Reading
The Fortress Language Specification

Masato Takeichi
IST, University of Tokyo

The Fortress Language Specification

- Version 1.0 Beta
  - March 6, 2007
- Available at

  http://research.sun.com/projects/plrg/fortress.pdf

Fortress Interpreter

- Available at

  http://fortress.sunsource.net

Chapter 1 Introduction (1)

- The Fortress Programming Language
  - General Purpose
  - Statically Typed
  - Component-based
- Designed for
  - Producing Robust High-performance Software
  - With Programmability

Chapter 1 Introduction (2)

- Fortress
  - Is a "Growable Language"
  - Supports state-of-the-art compiler optimization techniques
  - Has an extensible component system
  - Supports modular and extensible parsing
  - Name derived from Fortran
  - Has little relation to Fortran other than its intended application domain
  - Does not support for backward compatibility with existing versions of Fortran

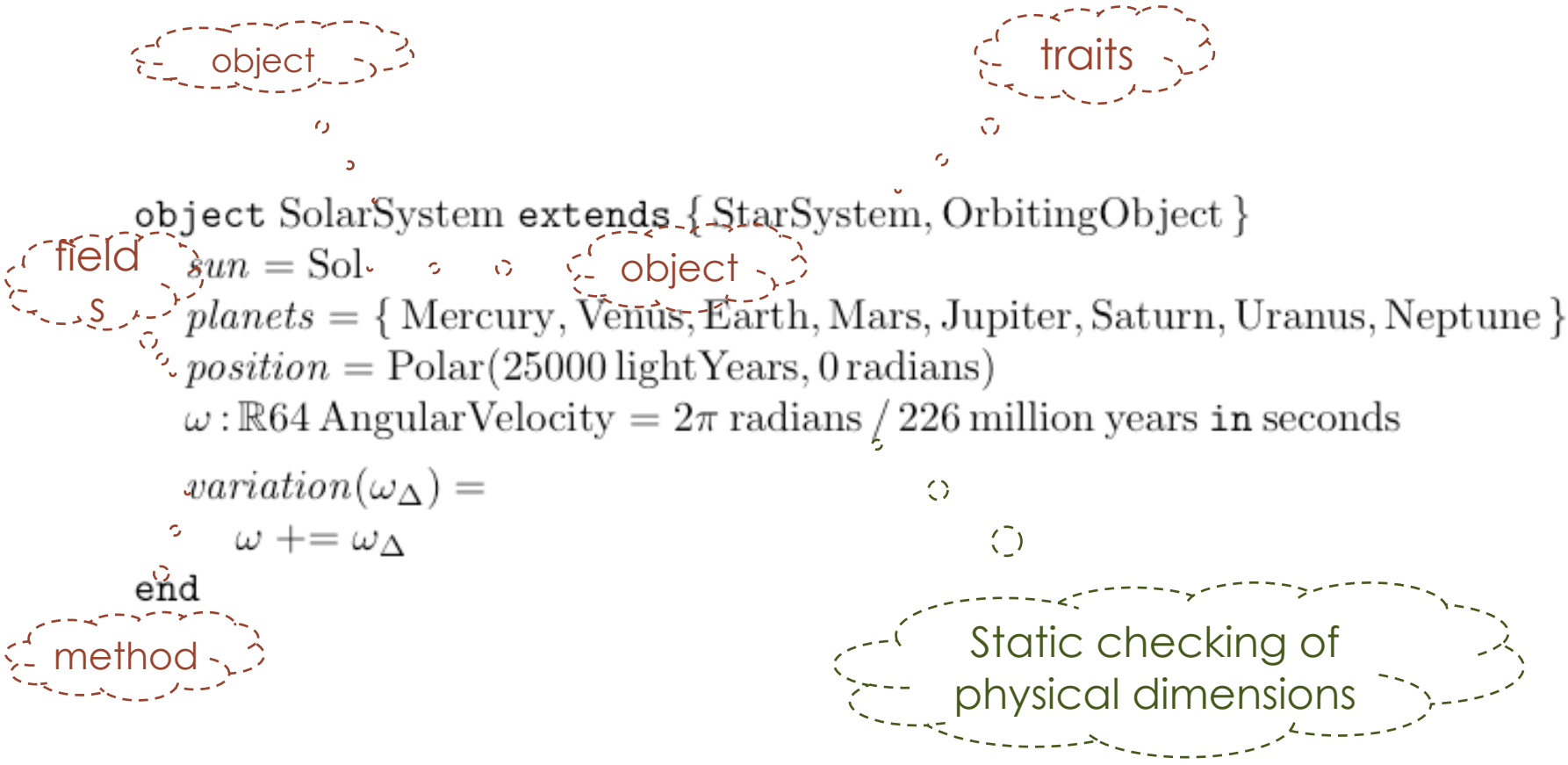
Chapter 1 Introduction (3)

- Aspects of Fortress
  - From object-oriented and functional languages
    - Java, NextGen, Scala, Eiffel, Self
    - Standard ML, Objective Caml, Haskell, Scheme
  - Employs cutting-edge features from programming language community
  - Achieves an unprecedented combination of performance and programmability
- Fortress is an "Open Source Project"

1.1 Fortress in a Nutshell (1)

- Object
  - Consists of *fields* and *methods* specified in definition
- Trait
  - Named program constructs that declare sets of methods
  - A method declared by trait may be either *abstract* or *concrete*
    - Abstract methods have *headers* only
    - Concrete methods also have *definitions*
  - May *extend* other traits
    - *Inherits* the methods provided by the traits it extends

1.1 Fortress in a Nutshell (2)



1.1 Fortress in a Nutshell (3)

- Notations
  - Allows Unicode characters, subscripts and superscripts in identifiers
  - Follows mathematical conventions
    - Variable references in *italics*
    - Multiplication expressed by simple juxtaposition
  - Supports operator overloading
  - Facilitates extension of syntax with domain-specific languages

1.1 Fortress in a Nutshell (4)

- Types
  - Statically and nominally typed
  - Types not specified for all fields, nor all method parameters and return values:
  - *Type inference* used wherever possible
  - Types can be parametric with respect to other types and values

1.1 Fortress in a Nutshell (5)

- Functions
  - Allows top-level function definitions in addition to objects and traits
  - Functions are *first-class* values:
    - Functions can be passed to and returned from functions
    - Functions are assigned as values to fields and variables
  - Functions and methods can be overloaded
  - Supports keyword parameters and variable size argument lists

1.1 Fortress in a Nutshell (6)

- Components
  - Programs are organized into *components*
  - Exports and Imports APIs and can be linked together
  - Component "Shape" described by APIs by specifying types in traits, objects and functions
  - External references are to APIs imported by component

1.1 Fortress in a Nutshell (7)

- Parallelism
  - Fortress supports a rich set of operations for defining parallel execution and distribution of large data structures
  - "For loops" are parallel by default

Chapter 2 Overview

2.1 The Fortress Programming Environment
2.2 Exports, Imports, and Linking Components
2.3 Automatic Generation of APIs
2.4 Rendering
2.5 Some Common Types in Fortress
2.6 Functions in Fortress
2.7 Some Common Expressions in Fortress
2.8 For Loops Are Parallel by Default
2.9 Atomic Expressions
2.10 Dimensions and Units
2.11 Aggregate Expressions
2.12 Comprehensions
2.13 Summations and Products
2.14 Tests and Properties
2.15 Objects and Traits
2.16 Features for Library Development

2.1 The Fortress Programming Environment (1)

- Fortress is platform independent
- A typical programming model:
  - Source code stored in files organized in directories
  - A text-based shell for
    - Store environment variables
    - Issue commands to execute and compile programs

2.1 The Fortress Programming Environment (2)

- Running a Program as a *script*
  - Program stored in a file with suffix ".fsx"

HelloWorld.fsx

```
export Executable
run(args) = print "Hello, world!"
```

  - Program executed directly from a shell by calling "fortress script" command

```
fortress script HelloWorld.fsx
```

2.1 The Fortress Programming Environment (3)

- Running a Program as a *compiled* code
  - Program stored in a file with suffix ".fss"
  - Program compiled into one or more *components* stored in a database *fortress*

HelloWorld.fss

```
component HelloWorld
    export Executable
    run(args) = print "Hello, world!"
end
```

  - Compile program by "fortress compile" command

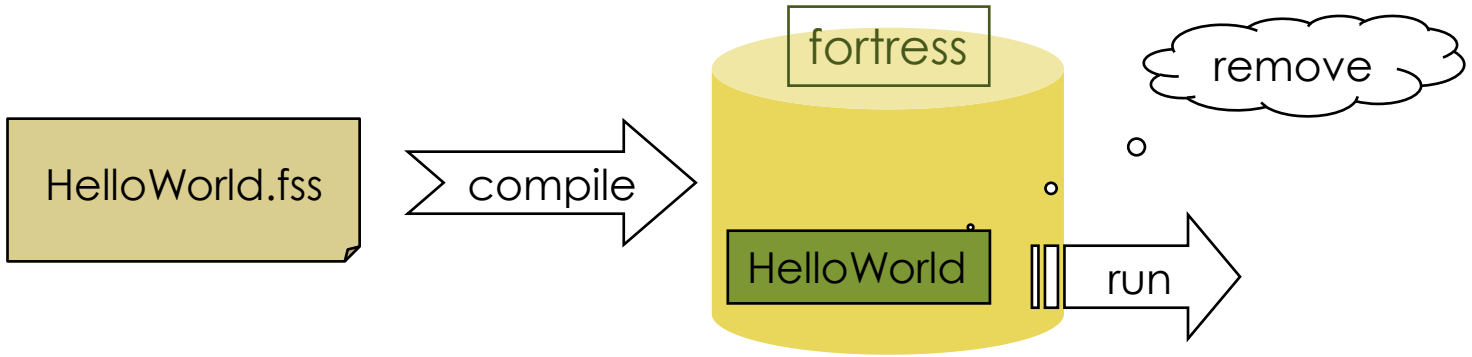
```
fortress compile HelloWorld.fss
```

  - Execute program by "fortress run" command

```
fortress run HelloWorld
```

2.1 The Fortress Programming Environment (4)

- Manipulating fortress
  - Components are stored in fortress by compilation
  - New component shadows the old one with the same name in fortress
  - Components are removed from fortress by "fortress remove" command



2.2 Exports, Imports, and Linking Components (1)

- Exporting API
  - Components can include *export statements*
    - Export statements list APIs that a components implement

```
component HelloWorld
    export Executable
    run(args) = print "Hello, world!"
end
```

Components implements *Executable* API

    - APIs are themselves program constructs

```
api Executable
    run: String... → ()
end
```

void

varargs parameter

2.2 Exports, Imports, and Linking Components (2)

- API
  - Defined in files with ".fsi"

Blarf.fsi

```
api Zeepf
    foo: String → ()
    baz: String → String
end
```

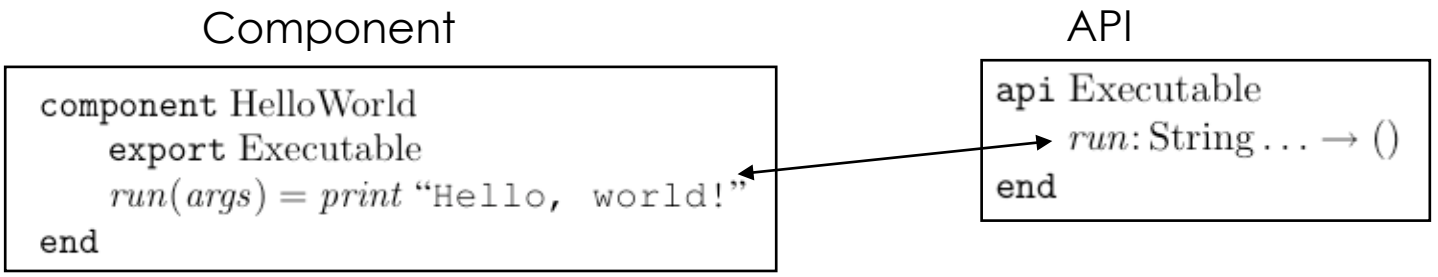
  - Compiled with "fortress compile" command

```
fortress compile Blarf.fsi
```

  - API compilation does not shadow existing elements of a fortress
    - error signaled if the same name exists.
  - API removed after all components referring to the API have been removed with "fortress removeAPI" command

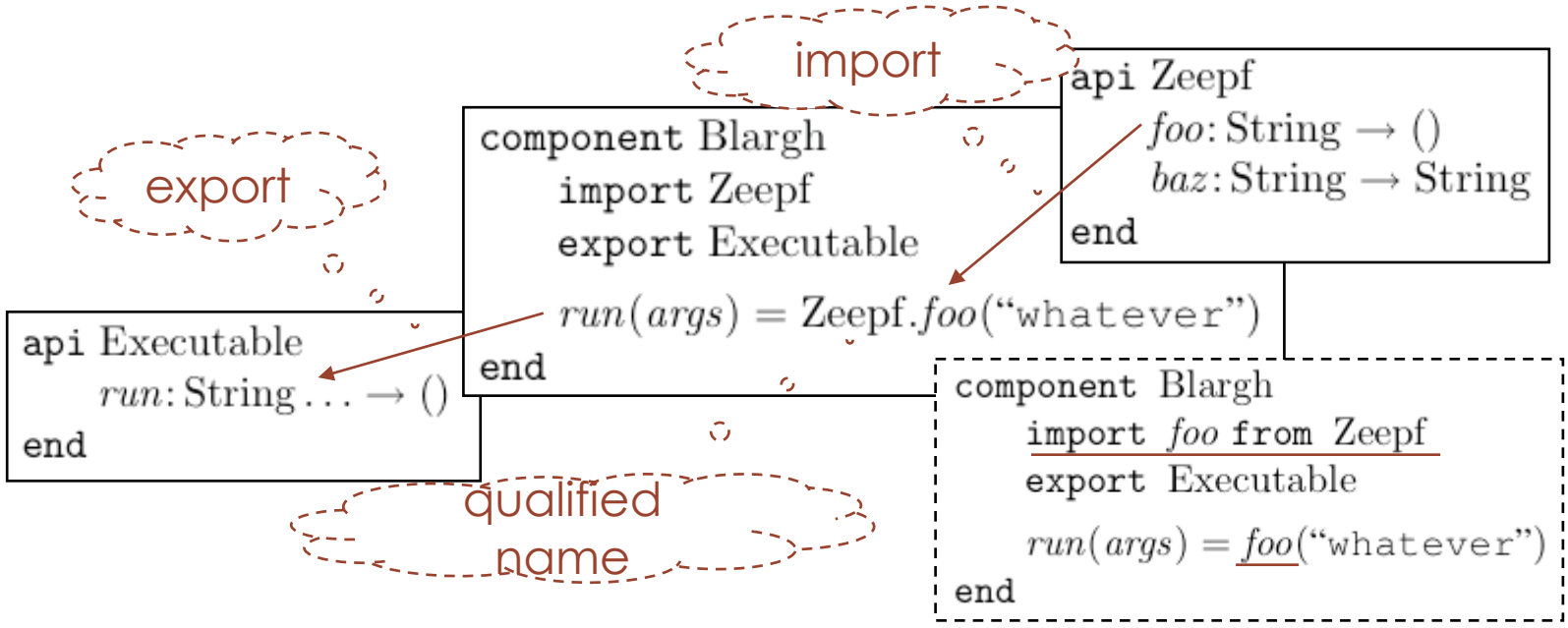
2.2 Exports, Imports, and Linking Components (3)

- Exporting API
  - Component exporting API must provide definition for every construct declared in that API



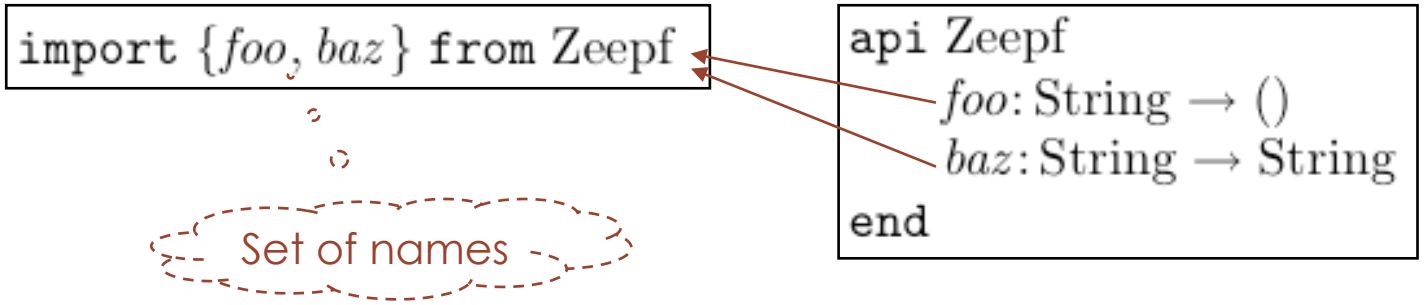
2.2 Exports, Imports, and Linking Components (4)

- Importing API (1)
  - Component importing API can use any constructs declared in that API



2.2 Exports, Imports, and Linking Components (5)

- Qualified Name
  - Import statement "import *S* from *A*" makes all names in set *S* imported from API *A*
  - Imported names can be referred to as *unqualified names* in that component



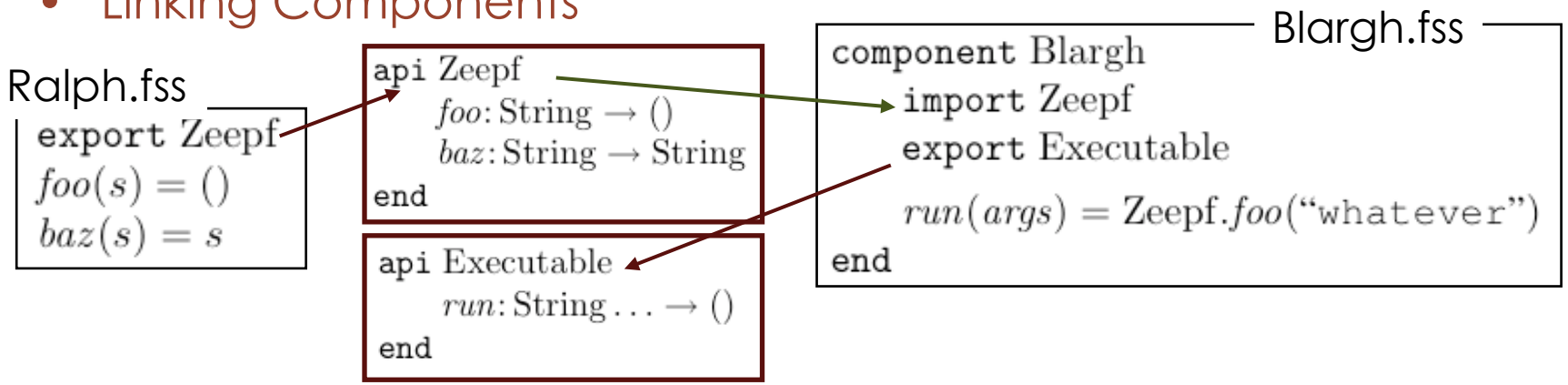
2.2 Exports, Imports, and Linking Components (6)

- Component Reference
  - No component refers directly to another component
  - *All external references go through APIs*

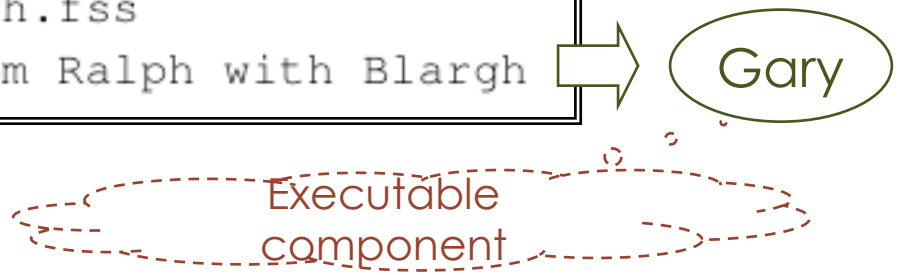
- Component
  - *Executable components*
    - contain no import statements
    - export the "API Executable"
  - Executable components are compiled and executed as stand-alone
  - Non-executable components must be compiled and linked with other components to form a new *compound* component

2.2 Exports, Imports, and Linking Components (7)

- Linking Components



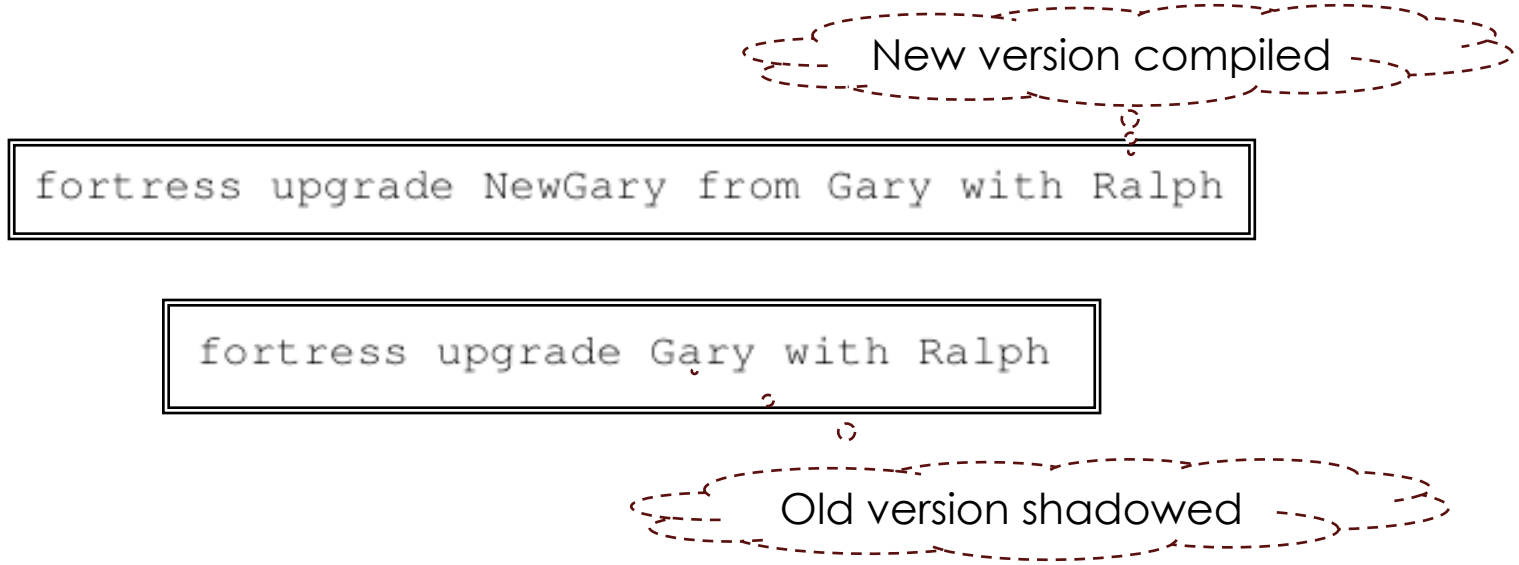

```
fortress compile Ralph.fss
fortress compile Blargh.fss
fortress link Gary from Ralph with Blargh
```



Execute with "fortress run Gary" command

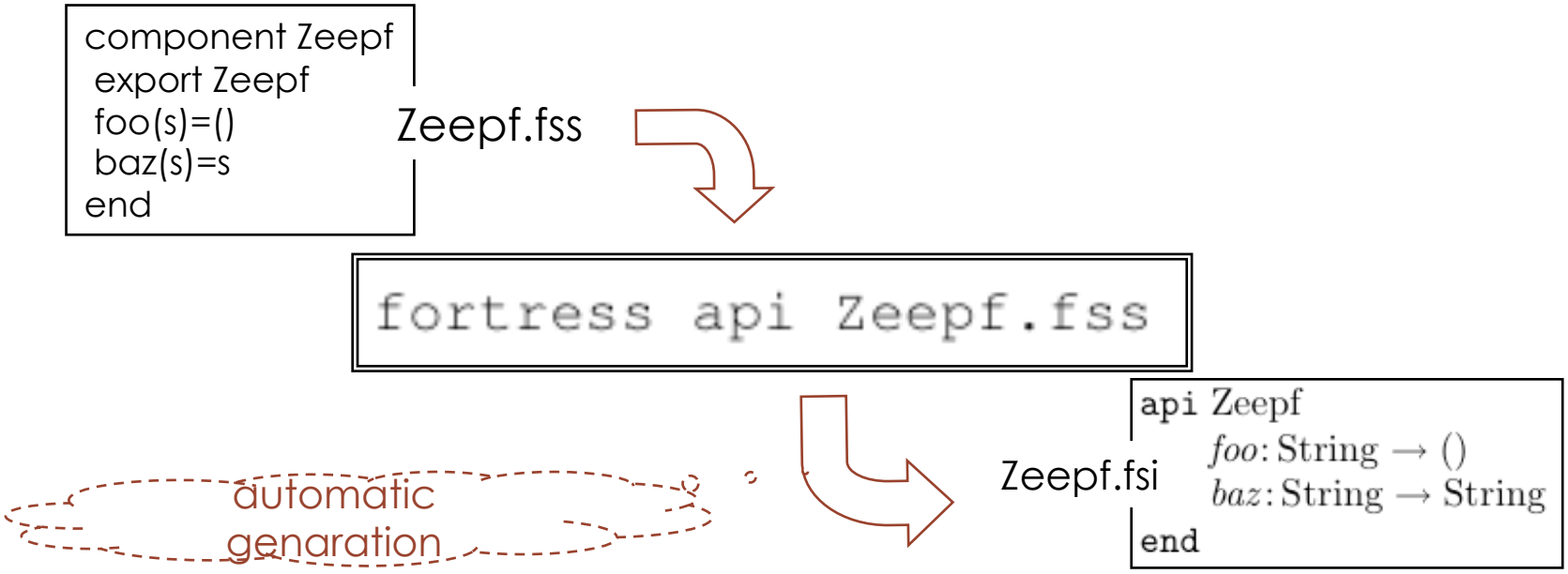
2.2 Exports, Imports, and Linking Components (8)

- Components are *encapsulated*
  - Compound components contain their own copies of constituent components in the resident fortress
- Compound components are *upgradable*
  - With new components that export some of the APIs used by theirconstituents



2.3 Automatic Generation of API

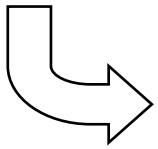
- Components and APIs exist in separate namespaces
  - Component may have same name as API



2.4 Rendering (1)

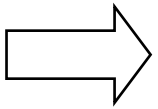
- ASCII representation is rendered as …

```
f(x) = x^2 + sin x - cos 2 x
```



$$f(x) = x^2 + \sin x - \cos 2x$$

```
a[i]
```



$$a_i$$

2.4 Rendering (2)

- ASCII names for Unicode characters

BY	*becomes*	×	TIMES	*becomes*	×
DOT	*becomes*	·	CROSS	*becomes*	×
CUP	*becomes*	∪	CAP	*becomes*	∩
BOTTOM	*becomes*	⊥	TOP	*becomes*	⊤
SUM	*becomes*	∑	PRODUCT	*becomes*	∏
INTEGRAL	*becomes*	∫	EMPTYSET	*becomes*	∅
SUBSET	*becomes*	⊂	NOTSUBSET	*becomes*	⊄
SUBSETEQ	*becomes*	⊆	NOTSUBSETEQ	*becomes*	⊈
EQUIV	*becomes*	≡	NOTEQUIV	*becomes*	≢
IN	*becomes*	∈	NOTIN	*becomes*	∉
LT	*becomes*	<	LE	*becomes*	≤
GT	*becomes*	>	GE	*becomes*	≥
EQ	*becomes*	=	NE	*becomes*	≠
AND	*becomes*	∧	OR	*becomes*	∨
NOT	*becomes*	¬	XOR	*becomes*	⊕
INF	*becomes*	∞	SQRT	*becomes*	√

2.5 Some Common Types in Fortress

- Standard type
  - String
  - Boolean
  - Numerics

64-bit precision float $\mathbb{R}$ (**RR** in ASCII) $\mathbb{R}64$

32-bit precision float $\mathbb{R}32$

64-bit integer $\mathbb{Z}64$

32-bit integer $\mathbb{Z}32$

Infinite precision integer $\mathbb{Z}$

2.6 Functions in Fortress (1)

- Allows (mutually) recursive function definitions

$$
\begin{array}{l}
factorial(n) = \\
\quad \texttt{if } n = 0 \texttt{ then } 1 \\
\quad \texttt{else } n\, factorial(n-1) \texttt{ end}
\end{array}
$$

Juxtaposition for multiplication

2.6 Functions in Fortress (2)

- 2.6.1 Juxtaposition and function application
  - Juxtaposed exprs of numeric type represent *multiplication*
    - *n factorial(n-1)*
  - Juxtaposition of expr of function type and another expr represents *function application*
    - sin x
  - Juxtaposition of expr of string type represents *concatenation*
    - “Hi,” “ it’s” “ me” “ again.”

2.6 Functions in Fortress (3)

- 2.6.2 Keyword Parameters

default value

$$makeColor(red : \mathbb{Z}64 = 0, green : \mathbb{Z}64 = 0, blue : \mathbb{Z}64 = 0) =$$
$$\texttt{if } 0 \le red \le 255 \wedge 0 \le green \le 255 \wedge 0 \le blue \le 255$$
$$\texttt{then } \mathrm{Color}(red, green, blue)$$
$$\texttt{else throw } \mathrm{Error} \texttt{ end}$$

chained boolean operator

throw expression

Calling $makeColor(green = 255)$ brings *red*=0 and *blue*=0

2.6 Functions in Fortress (4)

- 2.6.3 Varargs Parameters
  - Functions with variable number of arguments allowed

```
api Executable
    run: String ... → ()
end
```

void

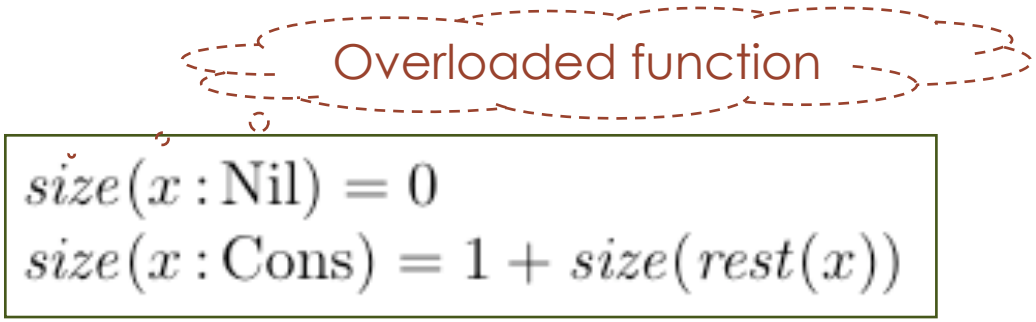
varargs parameter

```
printFirst(xs : ℝ ...) =
    if |xs| > 0 then print xs_0
    else throw Error end
```

testing the number of arguments

2.6 Functions in Fortress (5)

- 2.6.4 Function Overloading
  - Functions can be overloaded by parameter types
  - Overloaded calls resolved based on runtime types of arguments



2.6 Functions in Fortress (6)

- 2.6.5 Function Contracts

require contract

```
factorial(n) requires { n ≥ 0 }
    = if n = 0 then 1
      else n factorial(n − 1) end
```

```
factorial(n)
    requires { n ≥ 0 }
    ensures { result ≥ 0 }
    = if n = 0 then 1
      else n factorial(n − 1) end
```

ensure contract

2.7 Some Common Expressions in Fortress

while expression

```
while x < 10 do
    print x
    x += 1
end
```

block

```
printThreeWords() = do
    print "print"
    print "three"
    print "words"
end
```

tuple expression

```
("this","is","a","tuple","of","mostly","strings",0)
```

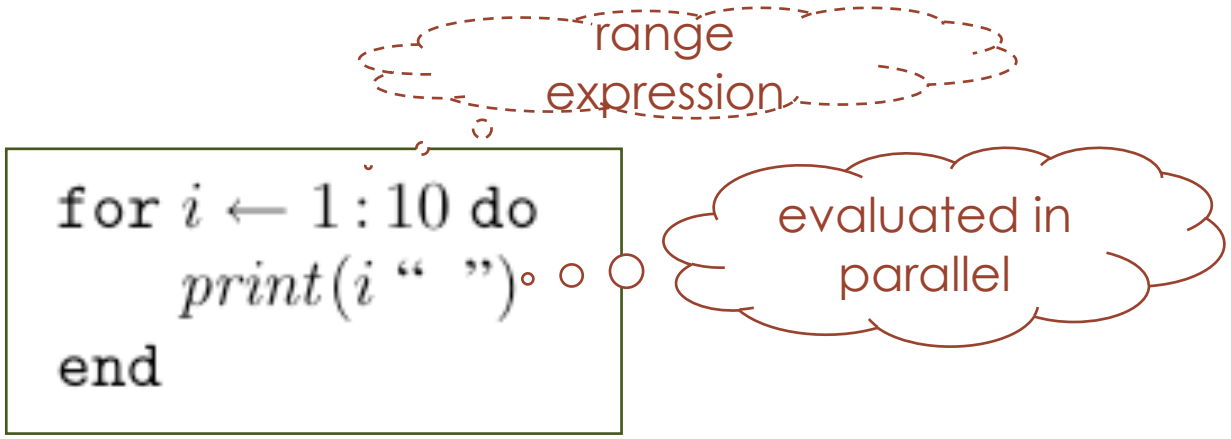
```
(factorial(100), factorial(500), factorial(1000))
```

⇔

```
do
    factorial(100)
also do
    factorial(500)
also do
    factorial(1000)
end
```

tuple elements evaluated *in parallel*

2.8 For Loops Are Parallel by Default



Result may be 5 4 6 3 7 2 9 10 1 8

2.9 Atomic Expressions (1)

- Atomic expression is executed in …
  - All other threads observe that
    - The computation has completed, or
    - The computation has not begun

```
atomic do
    x += 1
    y += 1
end
```

2.9 Atomic Expressions (2)

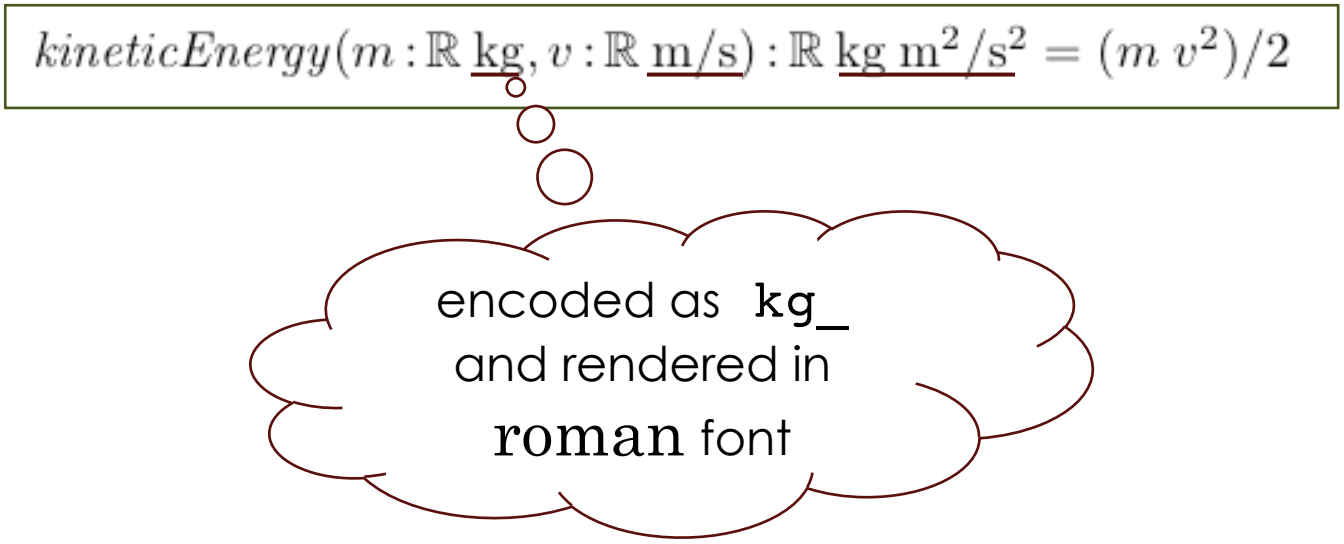
```
do
    x: ℤ := 0
    y: ℤ := 0
    z: ℤ := 0
    atomic do
        x += 1
        y += 1
    also atomic do
        z := x + y
    end
    z
end
```

This block observes that both x and y have been updated, or neither has

Possible values are 0 and 2

2.10 Dimensions and Units (1)

- Numeric types can be annotated with physical units and dimensions
  - Unit symbols are encoded with trailing underscores and rendered in roman font



2.10 Dimensions and Units (2)

- Longhand and shorthand names provided
  - m, meter, and meters
- Synonymous unit names provided
  - N is synonymous with kg m/s^2
  - Force is synonymous with Mass Acceleration
  - Acceleration is synonymous with Velocity/Time
- Measurements in the same unit can be …
  - compared, added, subtracted, multiplied, divided
- Measurements in different units can be …
  - multiplied, divided

2.10 Dimensions and Units (3)

Correct

$$v : \mathbb{R}\,\mathrm{m/s} = (3\,\mathrm{meters} + 4\,\mathrm{meters})/5\,\mathrm{seconds}$$

$$v : \mathbb{R}\,\mathrm{m/s} = (3\,\mathrm{meters} + 4\,\mathrm{seconds})/5\,\mathrm{seconds}$$

static error

$$v : \mathbb{R}\,\mathrm{m/s} = (3\,\mathrm{meters} + 4\,\mathrm{meters})/5$$

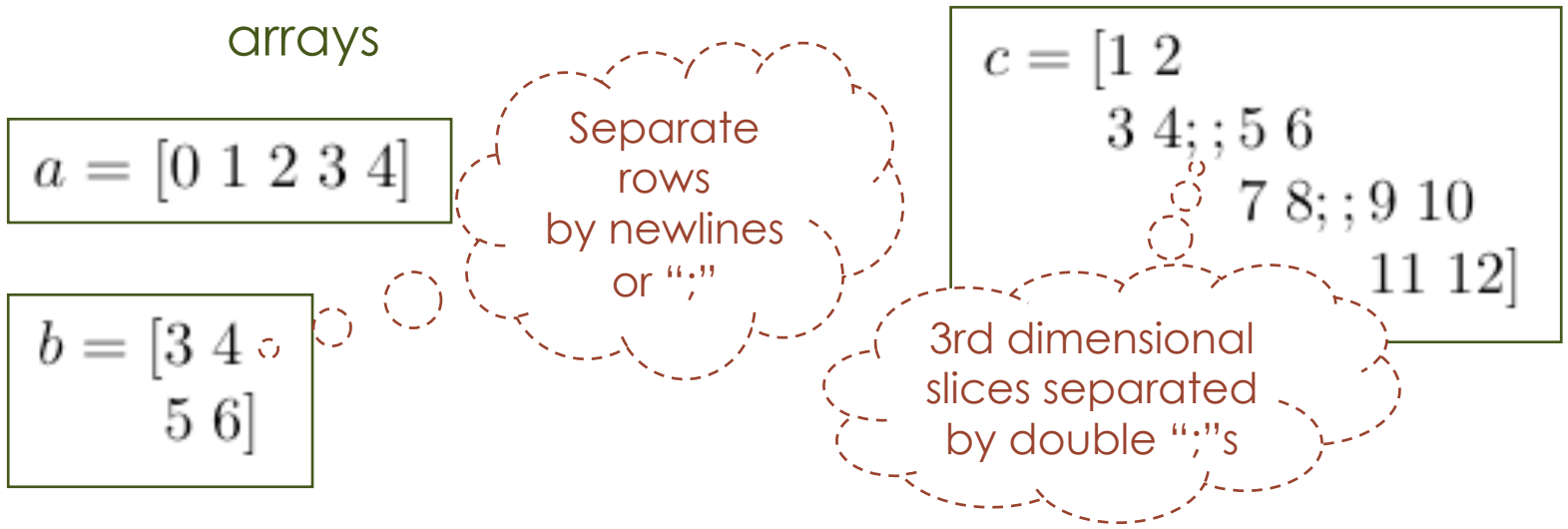
static error

$$kineticEnergy(3.14\,\mathrm{kg}, 32\,\mathrm{f/s}\ \mathbf{in}\ \mathrm{m/s})$$

unit conversion

2.11 Aggregate Expressions (1)

- Support for writing down collections by enumerating elements
  - Tuple, array, matrix, vector, map, set, list
- Elements are *evaluated in parallel*



2.11 Aggregate Expressions (2)

- Vector written down like one-dim array, Matrix written down like two-dim array
- Array aggregate expr evaluates to array, vector, or matrix from context
  - Elements of vectors and matrices must be numbers

$$[\,(\cos\theta)(-\sin\theta)\;\;(\sin\theta)(\cos\theta)\,]$$

Extra parentheses required

2.11 Aggregate Expressions (3)

- Set written down by …
  - Elements in braces, separated by commas
- List written down by …
  - Elements in angle brackets (written in ASCII as <|, |>)
- Map written down by …
  - Elements in braces with key/value pairs joined by arrow (written in ASCII as |->)

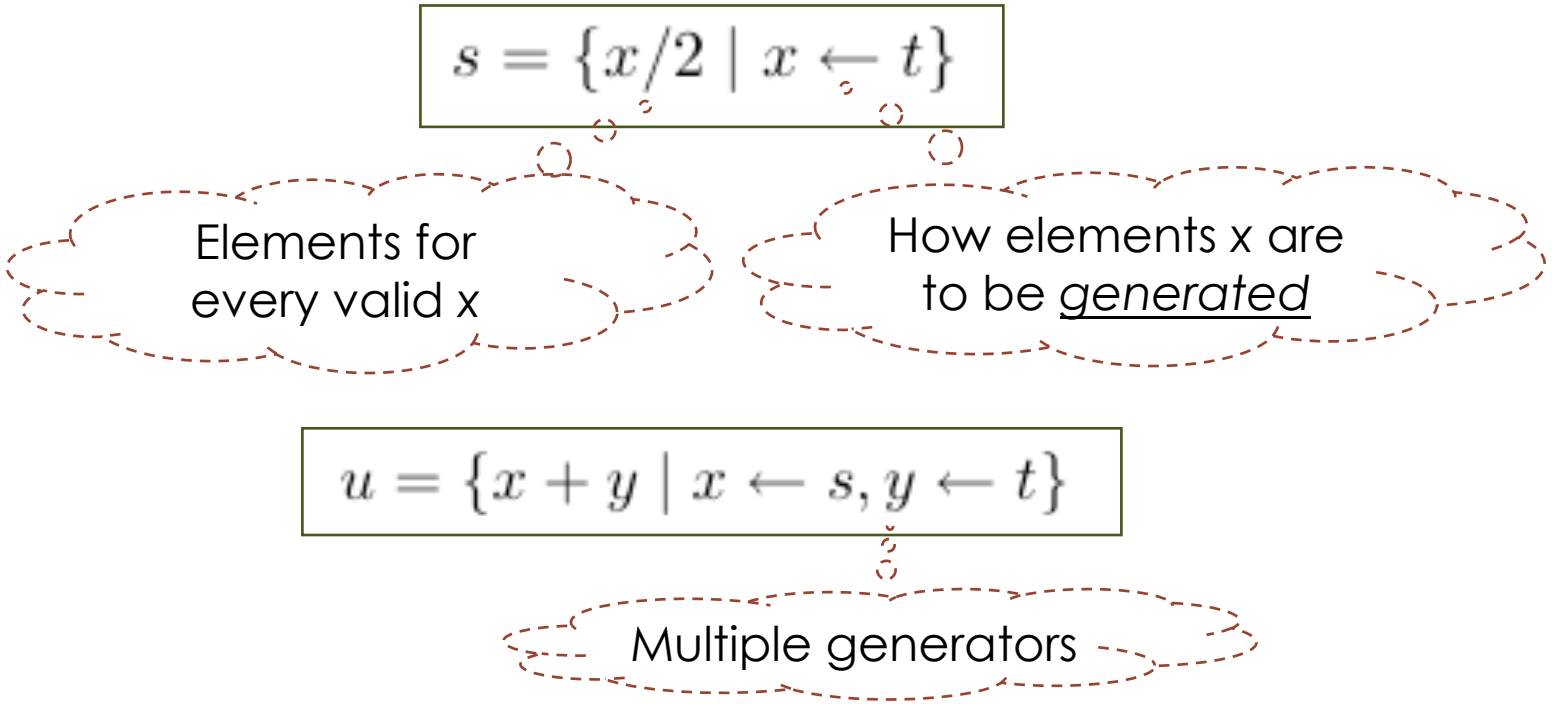
$$s = \{0, 1, 2, 3, 4\}$$

$$l = \langle 0, 1, 2, 3, 4 \rangle$$

$$m = \{'a' \mapsto 0, 'b' \mapsto 1, 'c' \mapsto 2\}$$

2.12 Comprehensions (1)

- Describe elements of collection by providing a rule



2.12 Comprehensions (2)

- Comprehension can contain *filtering expressions*
- Comprehension expression exists for aggregate except tuple

$$v = \{x \mid x \leftarrow t, x \geq 0\}$$

Filtering expression

List comprehension

$$\langle 2x \mid x \leftarrow v \rangle$$

2.12 Comprehensions (3)

- Array comprehension
  - Element expr includes a tuple indexing the elements of the array

$$[(x, y) = 0 \mid x \leftarrow \{0, 1, 2\}, y \leftarrow \{0, 1, 2\}]$$

```
[ 0 0 0
  0 0 0
  0 0 0 ]
```

Range expr 0..2 can be used

$$[(x, y) = 0 \mid x \leftarrow 0:2, y \leftarrow 0:2$$
$$(x, x) = 1 \mid x \leftarrow 0:2]$$

```
[ 1 0 0
  0 1 0
  0 0 1 ]
```

2.13 Summations and Products

- Syntactic support for "Big" operations

$$factorial(n) = \prod_{i \leftarrow 1:n} i$$

ASCII encoding

```
factorial(n) = PRODUCT[i <- 1:n] i
```

$\sum$ is written **SUM** in ASCII

2.14 Tests and Properties

- Support for automated program testing

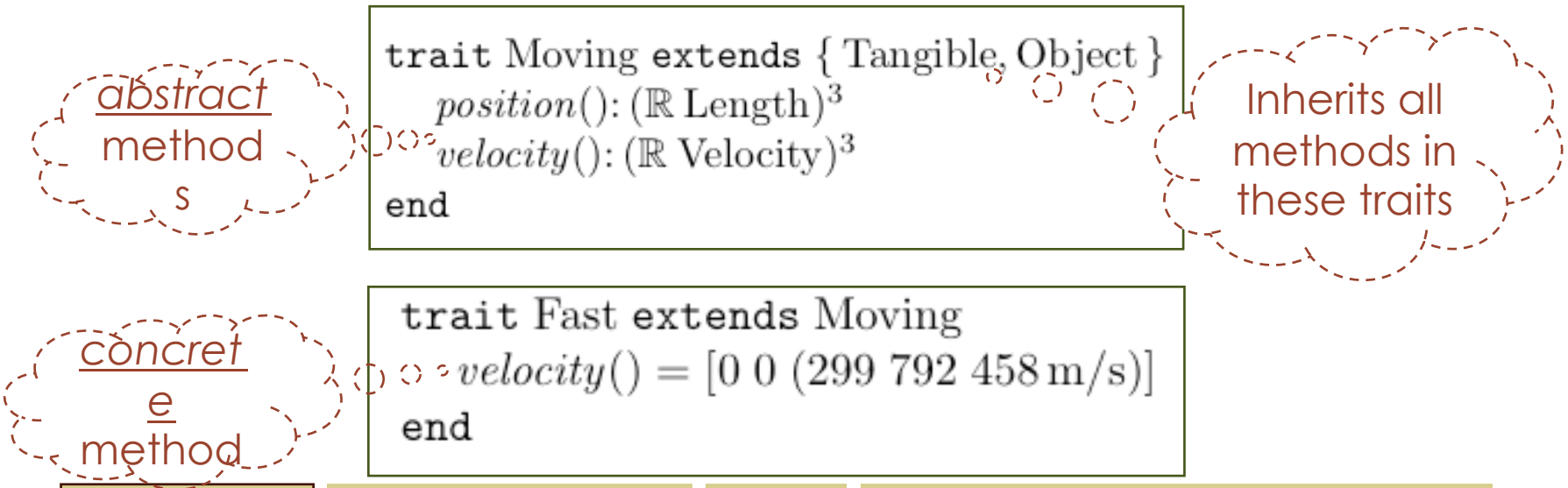
$$\texttt{test}\ factorialResultLarger[x \leftarrow 0:100] = x \leq factorial(x)$$

- Property declaration for documenting conditions expected
  - No explicit finite collections
  - Property expected to hold all values of the type

$$\texttt{property}\ factorialResultAlwaysLarger = \forall(x:\mathbb{Z})\ (x \leq factorial(x))$$

2.15 Objects and Traits (1)

- Defining new types as well as objects belonging to types
- Multiple inheritance hierarchy rooted at trait Object

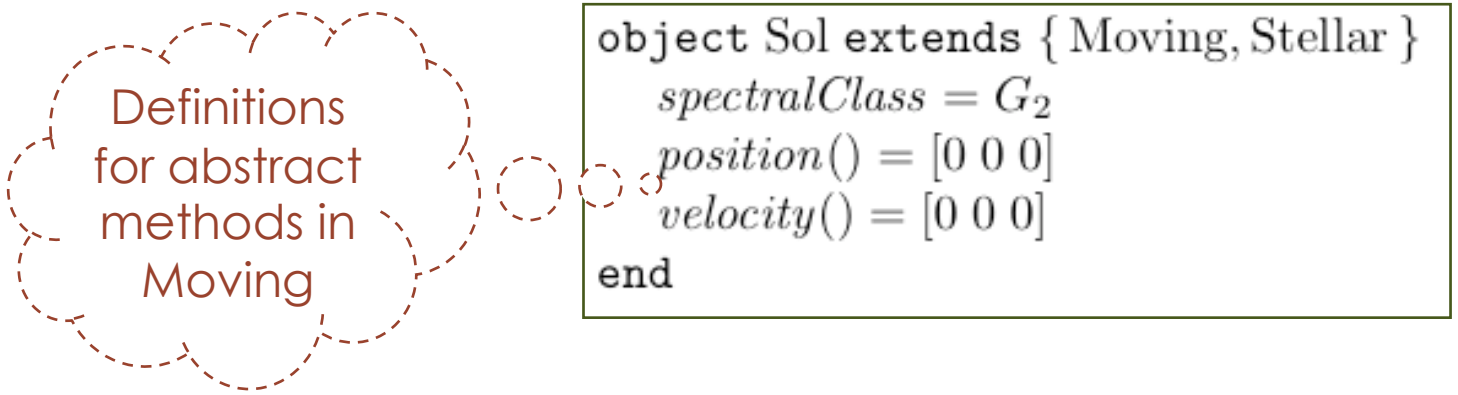


2.15 Objects and Traits (2)

- Trait declarations can be extended by …
  - trait declaration
  - *object declaration*
    - Singleton declaration declares a stand- alone *singleton object*
    - Constructor declaration declares an *object constructor*

2.15 Objects and Traits (3)

- Singleton declaration
  - Object must provide concrete definitions for all abstract methods it inherits



2.15 Objects and Traits (4)

- Fields can be declared in object definition
- For every field …
  - Implicit *getter* is defined
  - If a field includes modifier settable, implicit *setter* is defined

getter expr

$$\mathrm{Sol}.spectralClass$$

assignment

$$\mathrm{Sol}.spectralClass := G_3$$

2.15 Objects and Traits (5)

- Every method declared in an object or trait includes an implicit *self* parameter
  - self denotes the receiver of the method

```
object Sol extends {Moving, Stellar}
    spectralClass = G2
    self.position() = [0 0 0]
    velocity() = [0 0 0]
end
```

self param provided explicitly

```
trait List
    cons(x: Object, self) : List
    append(xs: List, self) : List
end
```

self params appear in nonstandard positions

2.15 Objects and Traits (6)

- Constructor declaration declares an *object constructor*
  - Declaration includes value param in header
- Every call to the constructor yields a new object

```
object Particle(position : (ℝ Length)^3,
                velocity : (ℝ Velocity)^3)
  extends Moving
end
```

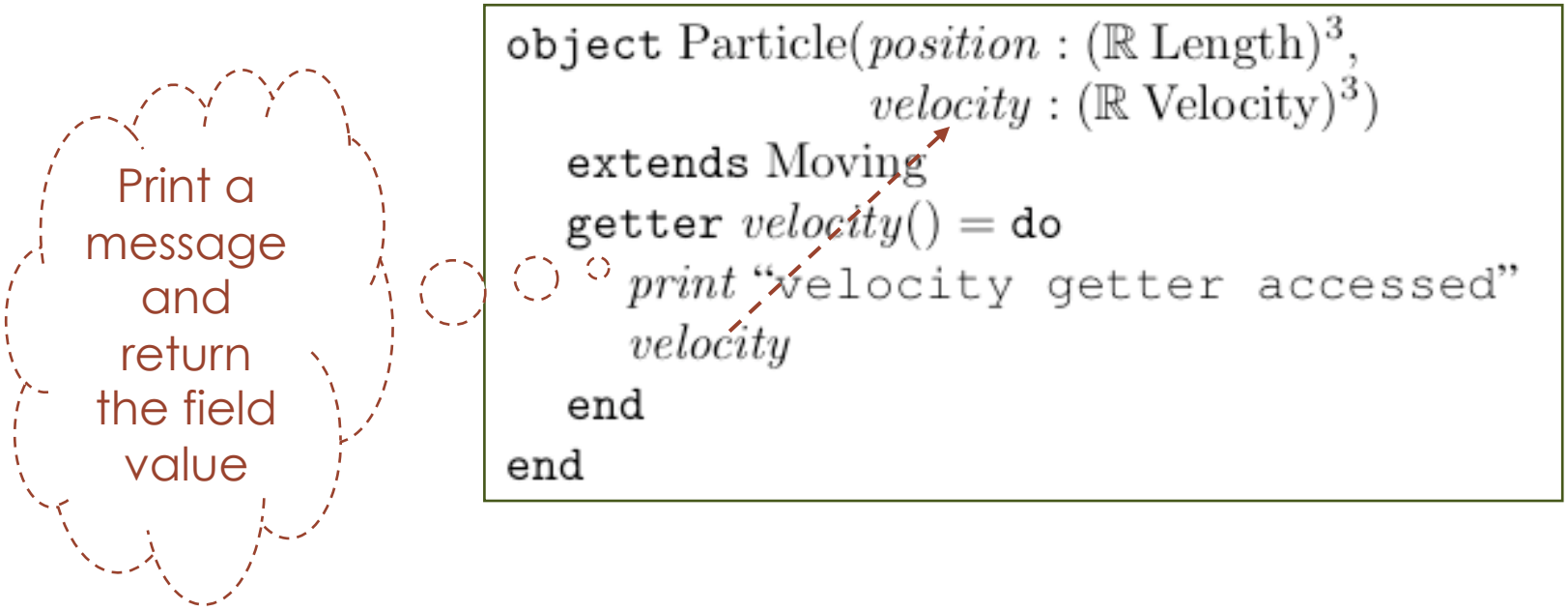
concrete definitions

```
p1 = Particle([3 2 5]m, [1 0 0]m/s)
```

Yields an object

2.15 Objects and Traits (7)

- Implicit getters and setters can be overridden



2.15 Objects and Traits (8)

- 2.15.1 Traits, Getters, and Setters
  - Traits do not include field declarations
  - Traits can include *getter* and *setter* declarations

getter decl

```
trait Moving extends { Tangible, Object }
    getter position(): (ℝ Length)^3
    getter velocity(): (ℝ Velocity)^3
end
```

```
v : Moving = ...
v.position
```

getter notation can be used for variable `v` of type `Moving`

2.15 Objects and Traits (9)

- Getters can be declared using field declaration syntax
- Getter declaration can include modifiers allowed on field declaration
  - settable is used for implicit *setter*

```
trait Moving extends { Tangible, Object }
    position: (ℝ Length)^3
    velocity: (ℝ Velocity)^3
end
```

Field decl syntax

2.16 Features for Library Development (1)

- Fortress designed to be a good language for *library* programming

- 2.16.1 Generic Types and Static Parameters
  - Allow types to be parametric such as Arrays and Vectors
  - Programmer can define new traits, objects, functions that include static parameters

2.16 Features for Library Development (2)

- 2.16.2 Specification of Locality and Data Distribution
  - Express programmer intent through data structure *distribution*
- 2.16.3 Operator Overloading

$$\texttt{opr}\ (n)! = \prod_{i \leftarrow 1:n} i$$

Postfix operator for factorial

2.16 Features for Library Development (3)

- 2.16.4 Definition of New Syntax
  - Provides a facility for defining new syntax in libraries

Example

Buffon's needle: Estimates pi using Monte Carlo simulation

```
component fortress.executable

export Executable

run(args:String...):()=do
   needleLength = 20
   numRows = 10
   tableHeight = needleLength numRows
   var hits : RR64 = 0.0
   var n : RR64 = 0.0

   println("Starting parallel Buffons")
   recordTime(6.0)
   for i <- 1#3000 do
      delta_X = random(2.0) - 1
      delta_Y = random(2.0) - 1
      rsq = delta_X^2 + delta_Y^2
      if 0 < rsq < 1 then
         y1 = tableHeight random(1.0)
         y2 = y1 + needleLength (delta_Y / sqrt(rsq))
         (y_L, y_H) = (y1 MIN y2, y1 MAX y2)
         if ceiling(y_L/needleLength) = floor(y_H/needleLength) then
                  atomic do hits += 1.0 end
         end
         atomic do n += 1.0 end
      end
   end
   probability = hits/n
   pi_est = 2.0/probability
   printTime(6.0)
   println("")
   print("estimated Pi = ")
   println(pi_est)
   end
end
```

Example

Buffon's needle (Rendered version)

```
run(args : String ...) : () = do
    needleLength = 20
    numRows = 10
    tableHeight = needleLength numRows
    var hits : ℝ64 = 0.0
    var n : ℝ64 = 0.0
    println("Starting parallel Buffons")
    recordTime(6.0)
    for i ← 1 # 3000 do
        δX = random(2.0) − 1
        δY = random(2.0) − 1
        rsq = δX^2 + δY^2
        if 0 < rsq < 1 then
            y1 = tableHeight random(1.0)
            y2 = y1 + needleLength(δY / sqrt(rsq))
            (yL, yH) = (y1 MIN y2, y1 MAX y2)
            if ceiling(yL / needleLength) = floor(yH / needleLength) then
                                atomic do hits += 1.0 end
            end
            atomic do n += 1.0 end
        end
    end
    probability = hits/n
    πest = 2.0/probability
    printTime(6.0)
    println("")
    print("estimated Pi = ")
    println(πest)
    end
end
```

Chapter 3 Programs (1)

- Program consists of Unicode 5.0 characters
  - May be rendered as subscripts, superscripts, italicized, …
- Program is *valid* if it satisfies all *static conditions*
  - Only valid programs can be *executed*
  - Validity must be checked before execution

Chapter 3 Programs (2)

- Executing valid program consists of *evaluating expressions*
  - Evaluation may modify *program state* yielding *result*
  - Result is a *value*, or an *abrupt completion*

- Characters of a valid program determine a sequence of *input elements*
- Input elements determine *program constructs*

Chapter 3 Programs (3)

- Program constructs may contain other program constructs
  - *declaration* and *expression*

- Semantics explained as …
  - Structure of input elements and program constructs with static constraints
  - How outcome of program execution is determined from sequence of constructs

Chapter 3 Programs (4)

- Programs are developed, compiled, and deployed as *encapsulated upgradable components* (Chapter 2.2)
- Fortress is …
  - block-structured
    - Program consists of nested *blocks* of code
    - Entire program is a single block
  - expression-oriented
    - "statements" are expression with type ( )
  - whitespace-sensitive

Chapter 4 Evaluation (1)

- State of executing program consists of a set of *threads* and a *memory*
- Communication with outside world through *input and output actions*
- Program execution consists of evaluating … in parallel
  - Body expression of *run* function
  - Initial-value expression of top-level variables and singleton object fields

Chapter 4 Evaluation (2)

- Threads evaluate expressions by taking *steps*
  - Step may *complete* the evaluation
    - No more steps possible, or
    - May result in an *intermediate expression*
- Dynamic program order: partial order among expressions
  - See Chapter 13

Chapter 4 Evaluation (3)

- Intermediate exprs are generalizations of Fortress exprs
  - Some cannot be written in programs
- Expr is *dynamically contained* within another expr
  - All steps for the first are taken between the beginning and completion of the second

4.1 Values (1)

- Value is the result of normal completion of an expr
- Value has …
  - *type*
  - *environment*
  - finite set of *fields*
- Every value is an object
  - *value object*
  - *reference object*
  - *function*

4.1 Values (2)

- Type specifies …
  - Names and types of its fields
  - Which names must be bound in its environment
  - Methods of the object
    - Only trait types have methods other than those inherited from type Any
- Fields …
  - In value object, each field is a value
  - In reference object, each field is a location
  - Functions and the value () have no fields

4.1 Values (3)

- Field has a name, which may be an identifier or an index
  - Only values of type LinearSequence or Heapsequence have fields named by indices (Sections 40.1 & 40.3)
- Field in value object is *immutable*
- Reference objects may have both *mutable* and *immutable* fields

4.1 Values (4)

- Values are constructed by …
  - Top-level function declaration and singleton declaration
  - Evaluating …
    - Object expression
    - Function expression
    - Local function declaration
    - Call to object constructor
    - Literal
    - Spawn expression
    - Aggregate expression
    - Comprehension

  with constructed value as the result of normal completion of evaluation

4.2 Normal and Abrupt Completion of Evaluation (1)

- Expression is evaluated until it *completes*
- Evaluation may …
  - *Complete normally* resulting in a value, or
  - *Complete abruptly*
- Abrupt completion has an associate value
  - Exception value thrown and uncaught
  - Exit value of an exit expression
- Exception …
  - Programmer-defined; thrown by a throw expression
  - Predefined exception; thrown by Fortress standard libraries, e.g., DivideByZeroException

4.2 Normal and Abrupt Completion of Evaluation (2)

- On abrupt completion…
  - Control passes to dynamically immediately enclosing expression
  - Until it is handled either by …
    - try expression if exception is being thrown, or
    - label expression if exit expr was evaluated
- If abrupt completion is not handled within a thread, thread itself completes abruptly
- If the main thread completes abruptly, program completes abruptly

4.3 Memory and Memory Operations (1)

- Memory: a set of abstract *locations*
  - Used to model sharing and mutation
- Location has an associated type and *contains* a value of that type
  - Type of value is a subtype of type of location
- Location can have non-object trait types; Value always has an object type
- Operations performed on memory …
  - Allocation
  - Read
  - Write

4.3 Memory and Memory Operations (2)

- Allocation creates a new location of a given type
- Allocation occurs when …
  - A mutable variable is declared
  - A reference object is constructed
    - A new location allocated for each field
- Locations are never reclaimed
  - In practice, reclaimed by garbage collection

4.3 Memory and Memory Operations (3)

- Allocated location afresh is *uninitialized*
- Fortress guarantees …
  - An *initializing write* performed if it is ever read
  - Initializing write occurs before any read
- Any location whose value …
  - can be written after initialization is *mutable*
  - cannot be written after initialization is *immutable*
- Mutable locations include
  - mutable variables
  - settable fields of a reference object
- Immutable locations include
  - Non-transient, non-settable fields

4.4 Threads and Parallelism (1)

- Two kinds of threads in Fortress …
  - *Implicit* threads
  - *Spawned* (*explicit*) threads
    - Objects created by spawn construct
- Thread may be in one of five states
  - Not started
  - Executing
  - Suspended
  - Normally completed
  - Abruptly completed

4.4 Threads and Parallelism (2)

- Every thread has a *body* and an *execution environment*
  - Body is an intermediate expression
  - Thread evaluates it in the context of execution environment
  - Both the body and the environment may change when the thread takes a step
- Execution environment is used to look up names in scope
  - Environment of newly created thread is that of the thread that created the new thread

4.4 Threads and Parallelism (3)
- Implicit thread

- A number of Fortress constructs are *implicitly parallel*
  - An implicitly parallel construct creates a *group* of implicit threads
- Implicitly parallel constructs …
  - Tuple expressions
    - Each element evaluated in a separate implicit thread
  - also do blocks
    - Each sub-block evaluated in a separate implicit thread

4.4 Threads and Parallelism (4)
- Implicit thread

- Implicitly parallel constructs … (Cont.)
  - Method invocations and function calls
    - Receiver/function and each argument evaluated in a separate implicit thread
  - for loops, comprehensions, sums, generated expressions, and big operators
    - Parallelism in loops specified by generators
    - Generators other than *sequential* generator execute each iteration in a separate implicit thread

4.4 Threads and Parallelism (5)
- Implicit thread

- Implicitly parallel constructs … (Cont.)
  - Extremum expressions
    - Each guarding expression evaluated in a separate implicit thread
  - Tests
    - Each test evaluated in a separate implicit thread

Extremum expression

```
case largest of
  1 mile ⇒ "miles are larger"
  1 kilometer ⇒ "we were wrong again"
end
```

4.4 Threads and Parallelism (6)

- Implicit thread

- Implicit threads run "fork-join" style
  - All threads in a group created together and must complete before the group completes
  - Programmer cannot single out implicit thread and operate upon it
  - Implicit threads need not be scheduled fairly

$$r : \mathbb{Z}64 := 0$$
$$(r := 1, \texttt{while}\ r = 0\ \texttt{do}\ \ \texttt{end})$$

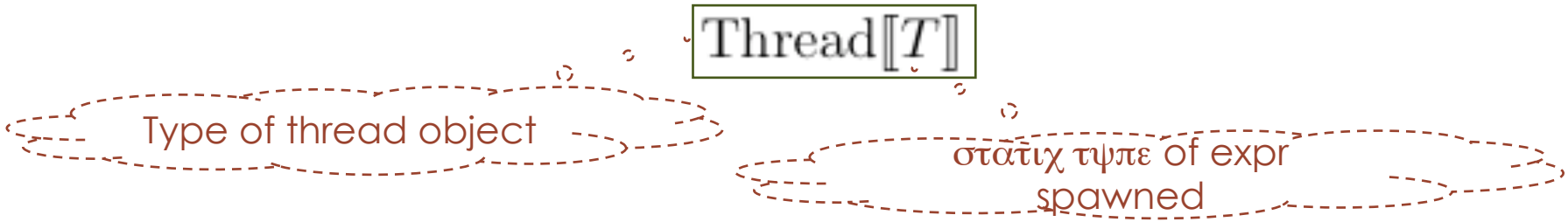
May loop forever

4.4 Threads and Parallelism (7)
- Implicit thread

- If any implicit thread completes abruptly, the group completes abruptly
  - Result of the group is the result of constituent thread that completes abruptly
  - Reduction variables should not be accessed after abrupt completion(Section 4.4.1)

4.4 Threads and Parallelism (8)
- Spawned thread

- Spawned thread objects are reference objects of …



- This trait has methods …
  - val returns value computed by spawn
    - Invocation of val may block until thread completes
  - wait waits for thread completion without return value
  - ready returns true if thread completes, false otherwise
  - stop attempts to terminate thread (Sec 32.6)

4.4 Threads and Parallelism (9)
- Spawned thread

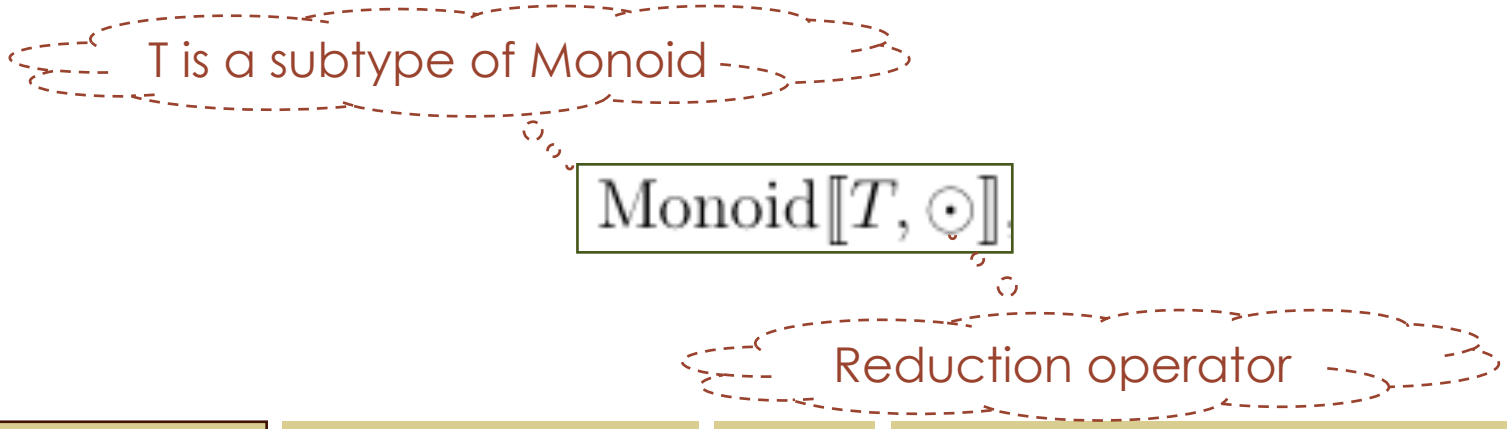
- Spawned thread has been *observed to complete* after invoking val or wait methods, or when ready invocation returns true
- In case of resource shortage, attempt made to run subexpression of spawn before continuing
  - The rest evaluated after the parallel block spawned off

4.4 Threads and Parallelism (10)

- Thread can be suspended in …
  - Thread that creates thread group is suspended until that group has completed
  - Thread that invokes val or wait is suspended until the spawned thread completes
  - Invoking *abort* function within atomic expression may cause thread to suspend
- Threads can perform operations simultaneously on shared objects
  - atomic expression synchronizes data access

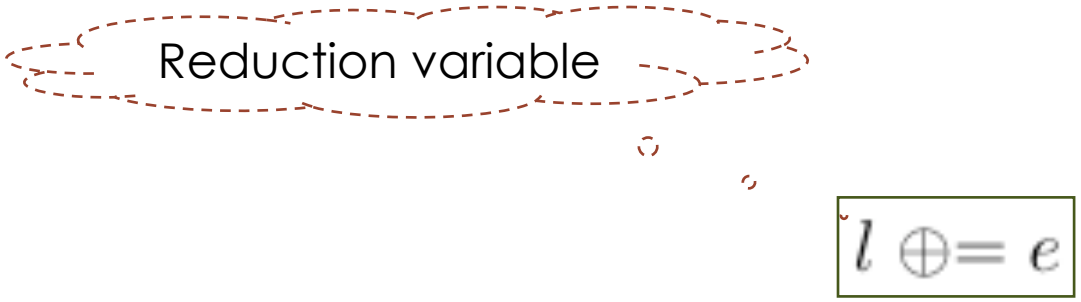
4.4.1 Reduction Variables (1)

- Special treatment to *reductions* for loops
- Reduction operator for type T is an operator on T generating a monoid
  - The operator is an associative infix on T



4.4.1 Reduction Variables (2)

- Reduction variable for thread group
  - Is of the form var op=expr using reduction operator or its group inverse
  - Value not read otherwise in the thread group
  - Variable is not a free variable of a functional



4.4.1 Reduction Variables (3)

- Other threads simultaneously reference a reduction variable see an arbitrary value
- Updates by those threads may be lost
- Association of terms in the reduction guided by loop generators (Sec 32.8)
- Fortress libraries declare common math operator to be monoid

Monoid operators

$+$, $\cdot$, $\wedge$, $\vee$, and $\underline{\vee}$.

$x \ominus= y$ also allowed for $\mathrm{Group}[\![T, \oplus, \ominus]\!]$

4.4.1 Reduction Variables (4)

- Implementation of reduction
  - Reduction var is assigned *identity* of reduction operator at beginning of iteration
  - When all iteration are complete, initial value and value of implicit thread are reduced and assigned to reduction var

Reductio
n variable

```
arraySum⟦nat x⟧(a : ℤ64[x]) : ℤ64 = do
    sum : ℤ64 := 0
    for i ← a.indices do
        sum += a[i]
    end
    sum
end
```

4.4.1 Reduction Variables (5)

- *Parallel slack* :

  (available work)/(number of threads)
  - Slack in hundreds or more proves beneficial with support for lightweight threading
  - Very slack computations easily adapt to differences in the number of processors
  - Slack is a desirable property

4.5 Environments (1)

- Environments maps *names* to values or locations
  - Environment is immutable
- Program starts with with empty environment
- Environments extended with mappings by
  - Variable/function/object declarations
  - Function calls
- After initializing top-level variables and singleton objects, *top-level environment* is constructed

4.5 Environments (2)

- Environment of value determined by how it is *constructed*
  - For object/function expr and local function decl, env of the constructed value is the lexical env in which expr/decl was evaluated
  - For others, env of the constructed value is top-level env of component in which expr/decl occurs
- Env of spawned thread for body is distinct from env of associated thread object in which calls to thread method are evaluated

4.6 Input and Output Actions

- Certain functionals (functions or methods) perform primitive input/output actions
- Any functional which may perform I/O action directly/indirectly must be declared with io modifier

Chapter 5 Lexical Structure

- Program consists of Unicode 5.0 characters
  - Every character is part of an input element
  - Partitioning of char sequence into input elements determined uniquely
- Standard ways to *render* (*display*) input elements described

5.1 Characters (1)

Code point

- *special non-operator characters*, which are:

U+0026	AMPERSAND	&	U+0027	APOSTROPHE	'
U+0028	LEFT PARENTHESIS	(	U+0029	RIGHT PARENTHESIS	)
U+002C	COMMA	,	U+002E	FULL STOP	.
U+0038	SEMICOLON	;	U+005C	REVERSE SOLIDUS	\
U+2026	HORIZONTAL ELLIPSIS	...	U+21A6	RIGHTWARDS ARROW FROM BAR	↦
U+2200	FOR ALL	∀	U+2203	THERE EXISTS	∃
U+2254	COLON EQUALS	:=			
U+27E6	MATHEMATICAL LEFT WHITE SQUARE BRACKET	⟦			
U+27E7	MATHEMATICAL RIGHT WHITE SQUARE BRACKET	⟧			

- *special operator characters*, which are

U+003A	COLON	:	U+003D	EQUALS SIGN	=
U+005B	LEFT SQUARE BRACKET	[	U+005D	RIGHT SQUARE BRACKET	]
U+005E	CIRCUMFLEX ACCENT	^	U+007B	LEFT CURLY BRACKET	{
U+007C	VERTICAL LINE	|	U+007D	RIGHT CURLY BRACKET	}
U+2192	RIGHTWARDS ARROW	→	U+21D2	RIGHTWARDS DOUBLE ARROW	⇒

- *letters*, which are characters with Unicode general category Lu, Ll, Lt, Lm or Lo—those with Unicode general category Lu are *uppercase letters*—and the following *honorary letters*:

U+221E	INFINITY	∞	U+22A4	DOWN TACK	⊤	U+22A5	UP TACK	⊥

- *connecting punctuation* (Unicode general category Pc);
- *digits* (Unicode general category Nd);

5.1 Characters (2)

- *prime characters*, which are:

U+2032	PRIME	U+2033	DOUBLE PRIME
U+2034	TRIPLE PRIME	U+2035	REVERSED PRIME
U+2036	REVERSED DOUBLE PRIME	U+2037	REVERSED TRIPLE PRIME

- *whitespace characters*, which are *spaces* (Unicode general category Zs) and the following characters:

U+0009	CHARACTER TABULATION	U+000A	LINE FEED
U+000B	LINE TABULATION	U+000C	FORM FEED
U+000D	CARRIAGE RETURN		
U+001C	INFORMATION SEPARATOR FOUR	U+001D	INFORMATION SEPARATOR THREE
U+001E	INFORMATION SEPARATOR TWO	U+001F	INFORMATION SEPARATOR ONE
U+2028	LINE SEPARATOR	U+2029	PARAGRAPH SEPARATOR

- *character literal delimiters*, which are:

U+0060	GRAVE ACCENT	`
U+2018	LEFT SINGLE QUOTATION MARK	‘
U+2019	RIGHT SINGLE QUOTATION MARK	’

- *string literal delimiters*, which are:

U+0022	QUOTATION MARK	"
U+201C	LEFT DOUBLE QUOTATION MARK	“
U+201D	RIGHT DOUBLE QUOTATION MARK	”

5.1 Characters (3)

- *ordinary operator characters*, enumerated (along with the special operator characters) in Appendix F, which include the following characters with code points less than U+007F:

U+0021	EXCLAMATION MARK	!	U+0023	NUMBER SIGN	#
U+0024	DOLLAR SIGN	$	U+0025	PERCENT SIGN	%
U+002B	PLUS SIGN	+	U+002D	HYPHEN-MINUS	−
U+003F	QUESTION MARK	?	U+0040	COMMERCIAL AT	@
U+002A	ASTERISK	*	U+002F	SOLIDUS	/
U+003C	LESS-THAN SIGN	<	U+003E	GREATER-THAN SIGN	>
U+007E	TILDE	~			

and most (but not all) Unicode characters specified to be *mathematical operators* (i.e., characters with code points in the range 2200-22FF) are operators in Fortress.

5.1 Characters (4)

- *control characters* are those with Unicode general category Cc;
- *ASCII characters* are those with code points U+007F and below;
- *protected characters* are the ASCII characters, control characters, and string literal delimiters;
- *word characters* are letters, digits, connecting punctuation, prime characters, and apostrophe;
- *restricted-word characters* are ASCII letters, ASCII digits, and the underscore character (i.e., ASCII word characters other than apostrophe);
- *hexadecimal digits* are the digits and the ASCII letters `A`, `B`, `C`, `D`, `E`, `F`, `a`, `b`, `c`, `d`, `e` and `f`;
- the *digit-group separator* is `NARROW NO-BREAK SPACE` (U+202F);
- *operator characters* are special operator characters and ordinary operator characters;
- *special characters* are special non-operator characters and special operator characters;
- *enclosing characters* are the enclosing operator characters enumerated in Section F.1, left and right parenthesis characters, and mathematical left and right white square brackets;

5.1 Characters (5)

- Forbidden and Restricted Characters
  - No control characters allowed except whitespace characters and `SUBSTITUTE (U+001A, "control-Z")`
  - Control characters cause static error if appear outside comment

U+0009	CHARACTER TABULATION	U+000B	LINE TABULATION
U+001C	INFORMATION SEPARATOR FOUR	U+001D	INFORMATION SEPARATOR THREE
U+001E	INFORMATION SEPARATOR TWO	U+001F	INFORMATION SEPARATOR ONE

5.2 Words and Chunks

- *Chunk* is a nonempty contiguous subsequence of program
- *Word* is a maximal chunk consising only word characters
  - Letters, digits, connecting punctuation, prime characters, apostrophe
- *Restricted word* is a maximal chunk with only restricted-word characters
  - ASCII letters, digits, underscore characters

5.3 Lines, Pages and Position

- Characters partitioned into *lines* and *pages*
- Line terminator
  - LINE FEED
  - CARRIAGE RETURN not immediately followed by LINE FEED
  - LINE SEPARATOR
  - PARAGRAPH SEPARATOR
- Page terminator
  - FORM FEED

5.4 ASCII Conversion

- An equivalent program containing only ASCII characters exists for every Fortress program
- ASCII conversion in three steps
  - Pasting words across line breaks
  - Replacing restricted words, sequences of operator/special characters with single Unicode characters
  - Replacing apostrophes in numerals with digit-group separators

5.5 Input Elements and Scannning

- ASCII conversion is followed by *scanning*
  - Program partitioned into *input elements*
- Input elements …
  - *Whitespace element* (*comments* included)
  - *Token*
    - *Reserved word*
    - *Literal*
      - Boolean, character, string, void, numeral
    - *Identifier*
    - *Operator token*
    - *Special token*

5.6 Comments

- Opening and closing comment delimiters …

  (* , *)

- Characters between balanced comment delimiters comprise a *comment*
  - Comment delimiters may be included in comments

5.7 Whitespace Elements

- Maximal chunk consisting of …
  - Comments
  - Whitespace characters not within string/ character literals, numerals
  - Ampersands not within string/character literals
- *Line-breaking whitespace* distinguished from *non-line-breaking whitespace*
- Static error if ampersand occurs unless …
  - Within character/string literal
  - Within comments
  - Immediately followed by line terminator or line-terminating comment

5.8 Reserved Words

BIG	SI_unit	absorbs	abstract	also	api	as
asif	at	atomic	bool	case	catch	coerces
coercion	component	comprises	default	dim	do	elif
else	end	ensures	except	excludes	exit	export
extends	finally	fn	for	forbid	from	getter
hidden	ident	if	import	in	int	invariant
io	juxtaposition	label	largest	nat	object	of
opr	or	private	property	provided	requires	self
settable	setter	smallest	spawn	syntax	test	then
throw	throws	trait	transient	try	tryatomic	type
typecase	unit	value	var	where	while	widening
widens	with	wrapped				

- Operators on units are also reserved
  - `cubed, cubic, in, inverse, per, square, squared`
- Reserved to avoid confusion (no special meaning in Fortress)
  - `goto, idiom, public, pure, reciprocal, static`

5.9 Character Literals

- Character literal consists of one or more characters enclosed in single quotation marks
  - ‘ … ’, ’ … ’, ` … ’

\b	U+0008	BACKSPACE
\t	U+0009	CHARACTER TABULATION
\n	U+000A	LINE FEED
\f	U+000C	FORM FEED
\r	U+000D	CARRIAGE RETURN
\"	U+0022	QUOTATION MARK
\\	U+005C	REVERSE SOLIDUS
\“	U+201C	LEFT DOUBLE QUOTATION MARK
\”	U+201D	RIGHT DOUBLE QUOTATION MARK

- Enclosed characters may be …
  - Single character
  - Sequence of hexadecimal digits for Unicode code point
  - Official Unicode 5.0 name/alternative name
  - ASCII characters converted to a Unicode character Character-literal escape sequence

5.10 String Literals

- String literal consists of sequence of characters enclosed in double quotation marks

5.11 Boolean Literals

- Boolean literals are false and true

5.12 The Void Literal

- Void literal is ( )

5.13 Numerals (1)

- Numeric literal (*numeral*) is a maximal chunk consisting one or more words satisfying …
  - Each word consists of only digits and letters
    - The last word may have one underscore as part of a *radix specifier*
  - Consecutive words separated by exactly one char, either a digit-group separator or '.'
  - The first word begins with a digit or the last word has a radix specifier

5.13 Numerals (2)

- *Radix specifier* is a word suffix consisting of
  - An underscore followed by
    - a sequence of one or more digits (interpreted in base 10), or
    - English name in all uppercase ASCII letters of an integer from 2 to 16

Valid numerals

```
17   7fff_16   0fff_SIXTEEN   10101101_2   XE_12   3.14159265   DEAD.BEEF_16
```

Both upper- and lowercase not allowed

Invalid numerals

```
0.a   0x6A35   FF_EIGHT   12.52.23   57_50   PI_FIFTEEN   dead.BEEF_16
```

5.14 Operator Tokens

- *Operator word* is …
  - Not reserved
  - Consists only of uppercase letters and underscores
  - Does not begin or end with underscore
  - Has at least two different letters
- *Base operator* is …
  - Ordinary operator character
  - Two-character sequence "**"
  - Sequence of two or more vertical-line char " | "
  - Multicharacter enclosing operator (Section 5.14.1)

5.15 Identifiers

- Word beginning with a letter and is not a reserved word, operator word, or all or part of a numeral

5.16 Special Tokens

- Every special character that is not part of a token

5.17 Rendering of Fortress Programs
5.17.1 Fonts

- Roman
- *Italic*
- *Math*
- *Script*
- **Fraktur**
- Sans-serif
- *Italic sans-serif*
- Monospace
- DOUBLE-STRUCK

5.17.2 Numerals

`27`	*is rendered as*	27
`7FFF_16`	*is rendered as*	$7FFF_{16}$
`10101101_TWO`	*is rendered as*	10101101_{TWO}
`37X8E2_12`	*is rendered as*	$37X8E2_{12}$
`deadbeef_SIXTEEN`	*is rendered as*	$deadbeef_{\mathrm{SIXTEEN}}$
`dead.beef_16`	*is rendered as*	$dead.beef_{16}$
`3.143159265`	*is rendered as*	3.143159265
`3.11037552_8`	*is rendered as*	3.1137552_{8}
`3.243f6b_16`	*is rendered as*	$3.243f6b_{16}$
`11.00100100001111111011010l0_2`	*is rendered as*	$11.00100100001111111011010l0_{2}$

5.17.3 Identifiers

- Complex rules for rendering identifiers …

Chapter 6 Declarations

- Declarations introduce *named entities*
  - Declaration *declares* an entity and a name
  - The declared name *refers to* the declared entity
- Not a one-one correspondence between declarations and named entities
- Declaration may contain other decls
  - Trait decl may contain method decls
  - Function decl may contain parameter decls

6.1 Kinds of Declarations (1)

Syntax:

Decl	::=	*TraitDecl*
	|	*ObjectDecl*
	|	*VarDecl*
	|	*FnDecl*
	|	*DimUnitDecl*
	|	*TypeAlias*
	|	*TestDecl*
	|	*PropertyDecl*

- Two kinds of declarations …
  - *Top-level declaration*
  - *Local declaration*

6.1 Kinds of Declarations (2)

- Top-level declarations …
  - Trait declarations (Chapter 9)
  - Object declarations (Chapter 10)
    - singleton decl/ constructor decl
  - Top-level variable declarations (Section 6.2)
  - Top-level function declarations (Chapter 12)/ top-level operator declarations (Chapter 16)
  - Dimension declarations (Chapter 18)
  - Unit declarations (Chapter 18)
  - Top-level type aliases (Section 8.9)
  - Test declarations (Chapter 19)
  - Top-level property declarations (Chapter 19)

6.1 Kinds of Declarations (3)

- Local declarations occur in another declaration or in some expression …
  - Field declarations (Section 10.2)
    - Occur in object decl and object expr
    - Include field decl in param list of constructor decl
  - Method declarations (Section 9.2)
    - Occur in trait/object decl, object expr
  - Coercion declarations (Chapter 17)
    - Occur in trait/object decl
  - Local variable declarations (Section 6.3)
    - Occur in block expr
  - Local function declarations (Section 6.4)
    - Occur in block expr

6.1 Kinds of Declarations (4)

- Local declarations occur in another declaration or in some expression … (cont'd)
  - Labeled blocks (Section 13.2)
  - Static-parameter declarations
    - Declare type param, nat param, int param, bool param, dim param, unit param, opr param, ident param
    - Occur in static-param lists of trait/object decl, top-level type aliases, top-level function decls, method decls
  - Hidden-type-variable declarations
    - Occur in where clauses of trait/object decls, top-level function decls, method decls
  - Type aliases in where clauses of trait/object decls, top-level function decls, method decls
  - (Value) parameter declarations
    - Occur in …

6.1 Kinds of Declarations (5)

- Some declarations are syntactic sugar for other decls
  - Apparent field decls in trait decls are method decls (Section 9.2)
  - Dimension/unit decl may desugar into several separate decls (Section 35.3)
  - After desugaring, the kinds of decls listed are disjoint

- Implicitly declared names
  - self implicitly declared as param of dotted methods (Section 9.2)
  - result implicitly declared as variable for ensures clause of a contract(Section 12.4)

6.1 Kinds of Declarations (6)

- *Type declarations* declare names that refer to *types*
  - Trait declarations
  - Object declarations
  - Top-level type aliases
  - Type-parameter declarations
  - Hidden-type-variable declarations
- *Dimension declarations*
  - Dimension declarations
  - dim-parameter declarations
- *Unit declarations*
  - Unit declarations
  - unit-parameter declarations

6.1 Kinds of Declarations (7)

- *Functional declarations*
  - Constructor declarations
  - Top-level function declarations
  - Method declarations
  - Local function declarations
- *Variable declarations*
  - Singleton declarations
  - Top-level variable declarations
  - Field declarations
  - Local variable declarations; incl implicit decl of result
  - (value) parameter declarations; incl implicit decl of self
- *Static-variable declarations*
  - Static-parameter declarations
  - Hidden-type-variable declarations

6.1 Kinds of Declarations (8)

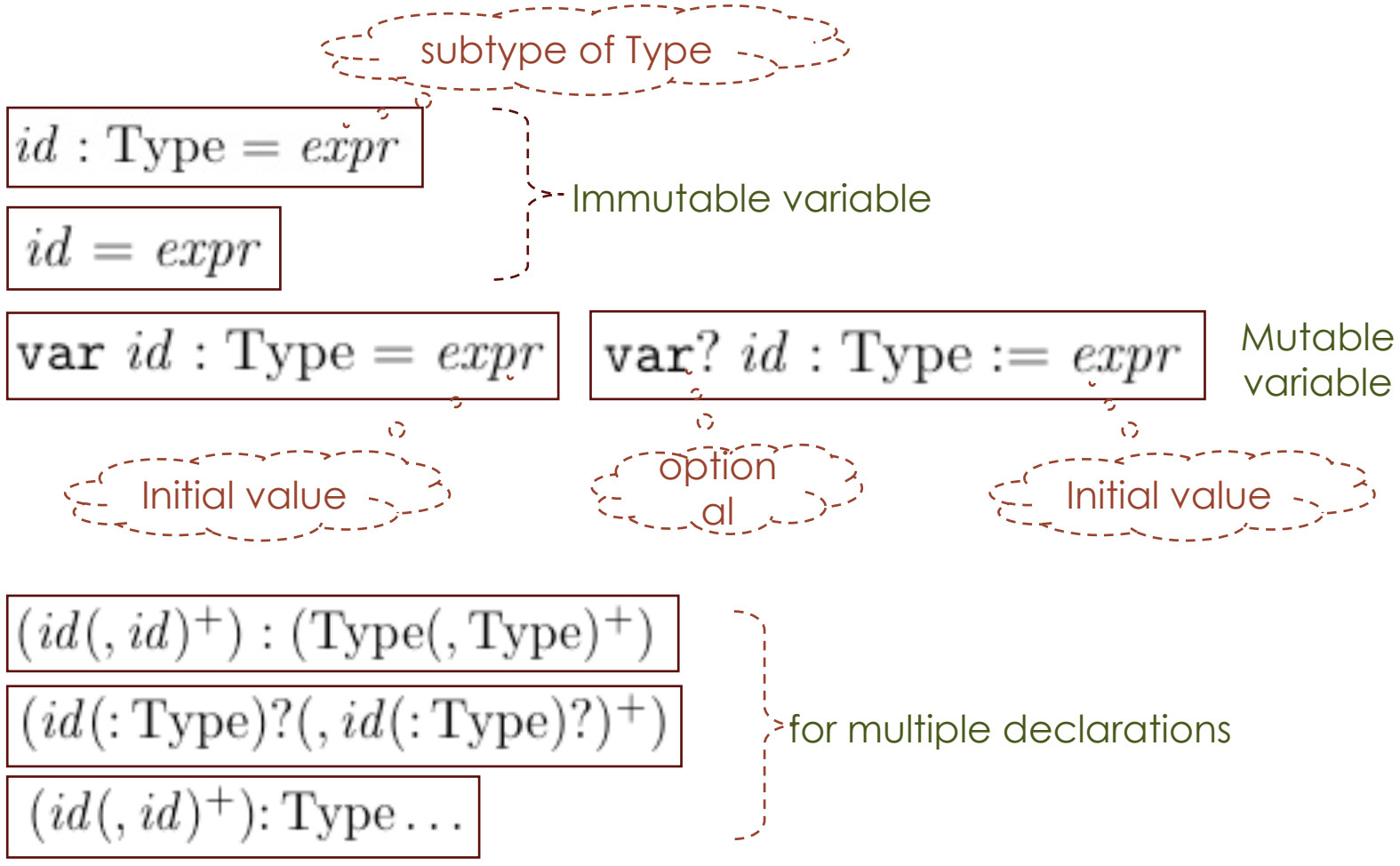
- Most declarations declare a single name given explicitly in the declaration
- One exception …
  - Wrapped field declarations (Section 9.3) in object decl and object expr
    - Declare both the field name and names for methods provided by the declared type of the field
- Method declarations in trait may be either *abstract* or *concrete*
  - Abstract decls do not have bodies
  - Concrete decls (called *definitions*) have bodies

6.2 Top-Level Variable Declarations (1)

Syntax:

VarDecl	::=	*VarWTypes InitVal*
	|	*VarWoTypes* = *Expr*
	|	*VarWoTypes* : *TypeRef* ... *InitVal*
	|	*VarWoTypes* : *SimpleTupleType InitVal*
VarWTypes	::=	*VarWType*
	|	(*VarWType*(, *VarWType*)$^+$)
VarWType	::=	*VarMods*? *Id IsType*
VarWoTypes	::=	*VarWoType*
	|	(*VarWoType*(, *VarWoType*)$^+$)
VarWoType	::=	*VarMods*? *Id*
InitVal	::=	(= | :=) *Expr*
SimpleTupleType	::=	(*TypeRef* , *TypeRefList*)
TypeRefList	::=	*TypeRef*(, *TypeRef*)*
VarMods	::=	*VarMod*$^+$
VarMod	::=	`var` | *UniversalMod*
IsType	::=	: *TypeRef*

6.2 Top-Level Variable Declarations (2)



6.2 Top-Level Variable Declarations (3)

Examples of variable declarations

$$\pi = 3.14159265358979323846264338327950288419716939937510820974944592307 8$$

$$\pi : \mathbb{R}64 = 3.14159265358979323846264338327950288419716939937510820974944592307 8$$

Example of multiple variable declaration using tuple notation

$$\texttt{var}\ (x, y) : \mathbb{Z}64 \ldots = (5, 6)$$

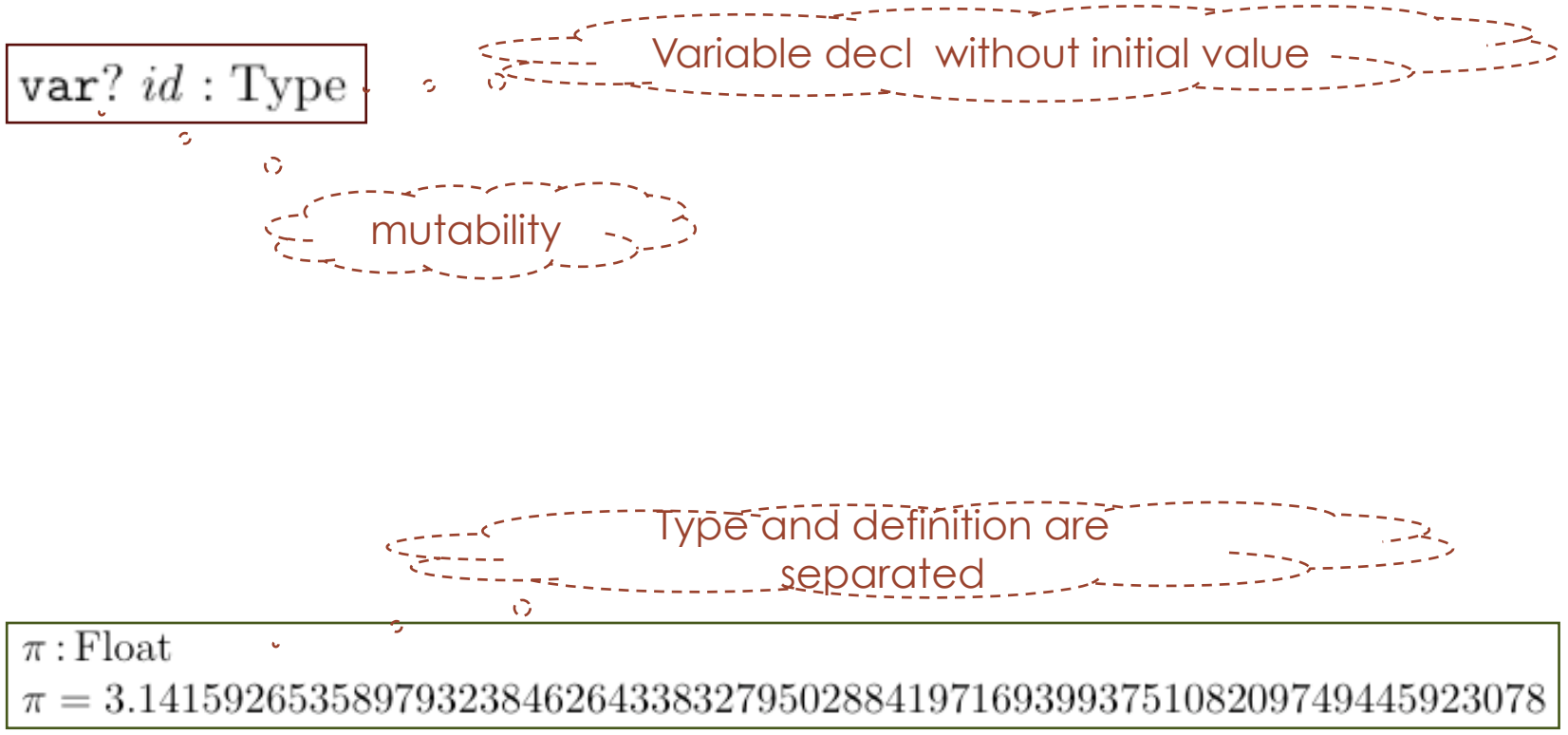
$$(x, y, z) : (\mathbb{Z}64, \mathbb{Z}64, \mathbb{Z}64) = (0, 1, 2)$$
$$(x : \mathbb{Z}64, y : \mathbb{Z}64, z : \mathbb{Z}64) = (0, 1, 2)$$
$$(x, y, z) : \mathbb{Z}64 \ldots = (0, 1, 2)$$

Equivalent declarations

6.3 Local Variable Declarations (1)

LocalVarDecl	::=	*LocalVarWTypes InitVal*
	|	*LocalVarWTypes*
	|	*LocalVarWoTypes* = *Expr*
	|	*LocalVarWoTypes* : *TypeRef* ... *InitVal*?
	|	*LocalVarWoTypes* : *SimpleTupleType InitVal*?
LocalVarWTypes	::=	*LocalVarWType*
	|	(*LocalVarWType*(, *LocalVarWType*)$^+$)
LocalVarWType	::=	`var` ? *Id IsType*
LocalVarWoTypes	::=	*LocalVarWoType*
	|	(*LocalVarWoType*(, *LocalVarWoType*)$^+$)
LocalVarWoType	::=	`var` ? *Id*
	|	*Unpasting*

6.3 Local Variable Declarations (2)



6.4 Local Function Declarations

LocalFnDecl	::=	*LocalFnMods*? *FnHeaderFront FnHeaderClause* = *Expr*
LocalFnMod	::=	`atomic` | `io`
LocalFnMods	::=	*LocalFnMod*$^{+}$

- Functions can be declared within block expressions
  - Via the same syntax as top-level func decls with modifiers private and test excluded

6.5 Matrix Unpasting (1)

Unpasting	::=	[*UnpastingElems*]
UnpastingElems	::=	*UnpastingElem RectSeparator UnpastingElems*
	|	*UnpastingElem*
UnpastingElem	::=	*Id* ([*UnpastingDim*])?
	|	*Unpasting*
UnpastingDim	::=	*ExtentRange* ($\times$ *ExtentRange*)$^+$
ExtentRange	::=	*StaticArg*? # *StaticArg*?
	|	*StaticArg*? : *StaticArg*?
	|	*StaticArg*
RectSeparator	::=	; +
	|	*Whitespace*

- Matrix unpasting is an extension of local variable declaration syntax as a shorthand for breaking a matrix into parts

6.5 Matrix Unpasting (2)

Cache-oblivious Matrix Multiplication

```
mm⟦nat m, nat n, nat p⟧(left : ℝ^(m×n), right : ℝ^(n×p), result : ℝ^(m×p)) : () =
  case largest of
    1 ⇒ result_(0,0) += (left_(0,0) right_(0,0))
    m ⇒ [ lefttop
          leftbottom ] = left
        [ resulttop
          resultbottom ] = result
        t_1 = spawn mm(lefttop, right, resulttop)
        mm(leftbottom, right, resultbottom)
        t_1.wait()
    p ⇒ [ rightleft rightright ] = right
        [ resultleft resultright ] = result
      t_1 = spawn mm(left, rightleft, resultleft)
      mm(left, rightright, resultright)
      t_1.wait()
    n ⇒ [ leftleft leftright ] = left
        [ righttop
          rightbottom ] = right
        mm(leftleft, righttop, result)
        mm(leftright, rightbottom, result)
  end
```

6.5 Matrix Unpasting (3)

Bind the Upper left square matrix to squareShape

```
foo⟦nat m, nat n⟧(A : ℝ^(m×n)) : () =
    if m < n then
        [squareShape_(m×m) rest] = A
        ...
    elif m > n then

        [squareShape_(n×n)
         rest] = A
        ...
    else (* A already square *)
        squareShape = A
        ...
    end
```

Chapter 7 Names

- Names used to refer to entities in Fortress program
- Names may be simple or qualified
  - Simple name: identifier or operator
  - Qualified name: API name followed by "." followed by an identifier
  - Operator cannot be qualified
- Simple names are introduced by declarations
  - Declaration may be implicit
  - Every declaration has a scope

7.1 Namespaces

- Fortress supports disjoint namespaces
  - Type namespace: Type declarations declare names and …
    - Static-variable declarations
    - Dimension declarations
  - Value namespace: Function and variable declarations declare names and …
    - nat, int, bool, unit, opr, ident parameters
  - Label namespace: names declared by labeled blocks

7.2 Reach and Scope of a Declaration (1)

- *Reach* of declaration
  - Reach of labeled block: the block itself
  - Reach of functional method declaration: component containing that declaration
  - Reach of dotted method declaration in trait T: declaration of T and any trait or object decl/ expr that extends T
  - Reach of other declaration: the smallest block strictly containing that declaration

7.2 Reach and Scope of a Declaration (2)

- If two declarations with overlapping reaches declare the same name in the same namespace, and the declarations are not overloaded, then one declaration *shadows* the other for that name in that namespace

7.2 Reach and Scope of a Declaration (3)

- Name is *in scope* in a namespace within the reach of any declaration that declares that name unless one of the conditions holds …
  - Declaration is shadowed at the program point for the name in that namespace
  - Declaration is a type alias, a dimension declaration, or unit declaration; program point is in the declaration
  - Declaration is a field, local variable or parameter declaration; program point is in the declaration or lexically precedes the declaration
  - Declaration is a parameter declaration of an object declaration; program point is in the body of a method declaration of that object declaration
  - Declaration is a labeled block; program point is in a spawn expression in the labeled block

7.3 Qualified Names

Syntax:

DottedId ::= *Id*(. *Id*)*

- Fortress provides a component system
  - Entities declared in a component are described by an API
- Component *imports* APIs
  - Allows to refer to entities declared by the imported APIs
  - In some cases, references to these entities must be *qualified* by the API name

Chapter 8 Types

- Fortress provides …
  - Trait types
  - Tuple types
  - Arrow types
  - BottomType
  - Other types provided in libraries
- Some types …
  - Have names
  - May be parameterized by types and values (*generic types*)
- Types are identical iff …
  - They are the same kind
  - Their names and arguments (if any) are identical

8.1 Relationships between Types (1)

- Types may be related by …
  - Subtyping relation
  - Exclusion relation
  - Coercion
- *Subtyping* relation is …
  - Reflexive, transitive, and antisymmetric
  - Defined by extends clause of trait and object decl and object expr

$T \preceq U$ ... *T* is a subtype of *U*

$T \prec U$ when $T \preceq U$ and $T \neq U$

8.1 Relationships between Types (2)

- Every expr has a *static type*
- Every value has a *runtime type* (dynamic type)
- Programs checked before executed to …
  - ensure the runtime type of the value is a subtype of the static type of the expr

- Fortress defines an *exclusion relation* between types which relates two disjoint types
  - No value can have a type that is a subtype of two types that exclude each other

8.1 Relationships between Types (2)

$T \lozenge U$

T excludes *U*

$T \lozenge U \implies \forall T' \preceq T : T' \lozenge U$

All subtypes excluded as well

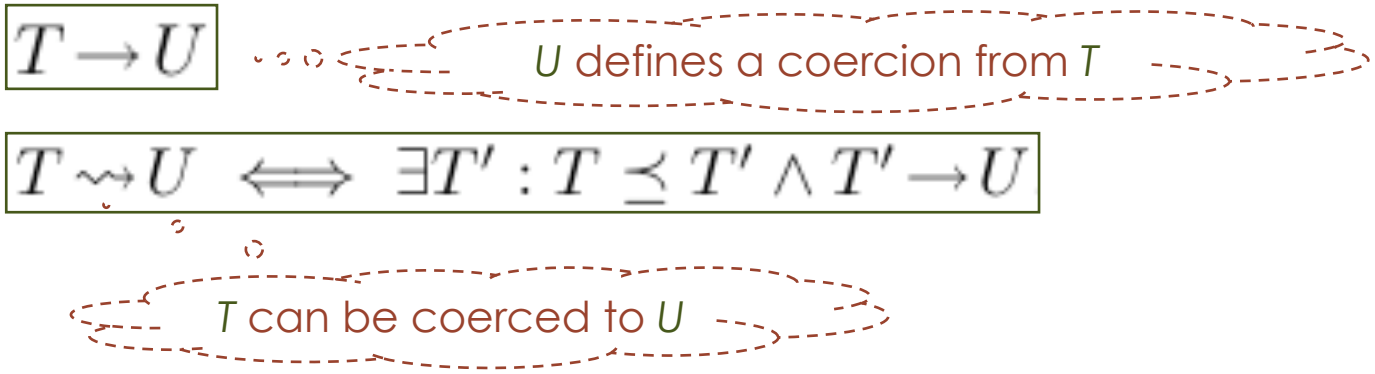
- Exclusion relation is …
  - Irreflexive and symmetric
  - Defined by excludes and comprises clauses of trait declarations
  - Implied from these by subtyping relation

```
trait S comprises {U, V} end
trait T comprises {V, W} end
object U extends S end
object V extends {S, T} end
object W extends T end
```

S and T excludes each other

8.1 Relationships between Types (3)

- *Coercion* between types
  - Coercion from *T* to *U* is defined in declaration of *U*



- Fortress type hierarchy is acyclic wrt subtyping and coercion except …
  - There exists a bidirectional coercion between two tuple types iff they have the same sorted form

8.2 Trait Types

Syntax:

TypeRef ::= *TraitType*

- Traits are declared by trait declarations (Chapter 9)
- Trait has a trait type of the same name

8.3 Object Trait Types

- Named objects are declared by object declarations (Chapter 10)
  - Named object has an *object trait type* of the same name
- Anonymous objects are declared by object expressions (Section 13.9)
  - Anonymous object has an anonymous object trait type
- Object trait type is a special kind of trait type

```
object Empty extends List
    first() = throw Error
    rest() = throw Error
    cons(x) = Cons(x, self)
    append(xs) = xs
end
```

```
object extends {List}
    first() = throw Error
    rest() = throw Error
    cons(x) = Cons(x, self)
    append(xs) = xs
end
```

8.4 Tuple Types (1)

TypeRef	::=	*TupleType*
TupleType	::=	((*TypeRef* ,)* (*TypeRef* ... ,)? *KeywordType*(, *KeywordType*)*)
	|	((*TypeRef* ,)* *TypeRef* ...)
	|	*SimpleTupleType*
KeywordType	::-	*Id* = *TypeRef*
SimpleTupleType	::=	(*TypeRef* , *TypeRefList*)

- Tuple is an ordered sequence of keyword-value pairs (Section 13.27)
- Tuple type consists of a parenthesized comma-separated list of …
  - A plain type *T*
  - A vararg type *T*...
  - A keyword-type pair *identifier=T*

8.4 Tuple Types (2)

- Element type in tuple type corresponds to one in tuple type iff …
  - Both are plain types in the same position
  - Both are vararg types, or
  - Both are keyword-type pairs with the same keyword
- Every tuple type is a subtype of Tuple
  - Tuple types are not subtypes of Object
  - Tuple types cannot be extended by other trait types

8.4 Tuple Types (3)

- Tuple types are covariant; tuple type *X* is a subtype of tuple type *Y* iff …
  - Correspondence between their element types is bijective
  - For each element type in *X*, the type in the element type is a subtype in the corresponding element type in *Y*
  - Keyword-type pairs in *X* and *Y* appear in the same order
- Sorted form *X'* for tuple type *X*
  - Created by reordering keyword-type pairs
  - There is a coercion from tuple type *X* to tuple type *Y* iff *X* and *Y* have the same sorted form

8.4 Tuple Types (4)

- Tuple type excludes any nontuple type other than Any
- Two tuple types exclude each other unless the correspondence between their element type is bijective
  - Two tuple types with bijective correspondence exclude each other if either any type of one excludes the type of the other, or their keyword-type pair do not appear in the same order
- Intersection of nonexclusive tuple types are defined elementwise

8.5 Arrow Types (1)

Syntax:

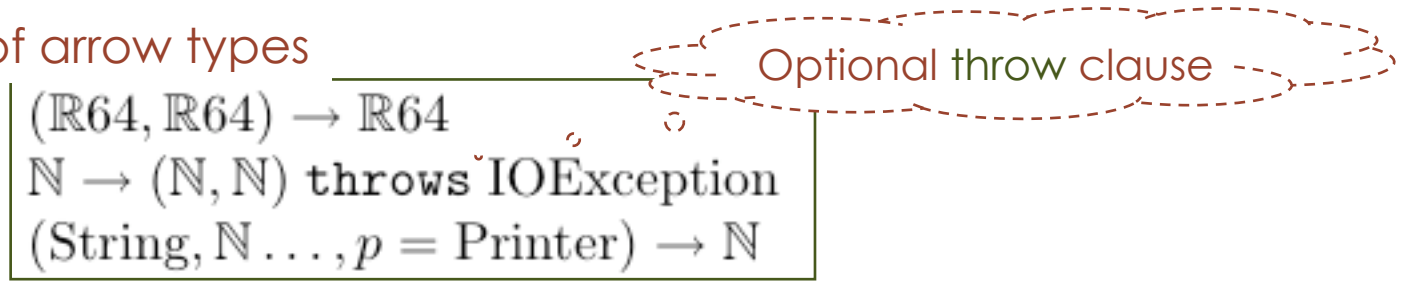
TypeRef ::= *ArrowType*

ArrowType ::= *TypeRef* → *TypeRef Throws?*

- *Arrow types*: types of function values
  - Functions can be passed as arguments and returned as values
- Every arrow type is a subtype of Object
- Arrow types are not trait types
  - Arrow types cannot be extended by other trait types

8.5 Arrow Types (2)

Examples of arrow types



- Parameter types are contravariant; return types are covariant

$A \to B$ `throws` C is a subtype of $D \to E$ `throws` F

iff

- D is a subtype of A and
- B is a subtype of E and
- For all X in C, there exists Y in F such that X is a subtype of Y

8.6 Bottom Type

- Fortress provides a special BottomType
- No value in Fortress has the bottom type
  - throw and exit expressions have the bottom type
- Bottom type is a subtype of every type
- Intersection of any exclusive types is the bottom type

8.7 Types in the Fortress Standard Libraries (1)

- Fortress standard libraries define simple standard types for literals (Section 13.1)
  - BooleanLiteral[b]
  - () (pronounced "void")
  - Character
  - String
  - Numeral[n,m,r,v]
  - Several simple numeric types
- Simple standard types for literals are mutually exclusive
- Values of these types are immutable

8.7 Types in the Fortress Standard Libraries (2)

- Numeric types share the common supertype Number
  - Arbitrary-precision integers Z
  - Unsigned arbitrary-precision integers N
  - Rational numbers Q
  - Fixed-size representation for integers
    - Z8, Z16, Z32, Z64, Z128
  - Fixed-size representation for unsigned integers
    - N8, N16, N32, N64, N128
  - Floating-point numbers
  - Intervals Interval[X]
    - X can be instantiated with any number type
  - Imaginary and complex numbers in …
    - rectangular form C*n* (n=16, 32, 64, 128, 256)
    - Polar form Polar[X] (X is instantiated with any real number type)

8.7 Types in the Fortress Standard Libraries (3)

- Floating-point numbers
  - R32, R64 to be 32 and 64-bit IEEE754 floating-point numbers
- Two functions on types …
  - Double[F] is a floating-point type twice the size of floating-point type F
  - Extended[F] is a floating-point type sufficiently larger than floating-point type F to perform summations

8.7 Types in the Fortress Standard Libraries (4)

- Fortress standard libraries also define …
  - Any
  - Object
  - Exception
  - Boolean
  - BooleanInterval
  - LenearSequence
  - HeapSequence
  - BinaryWord
  - …

8.8 Intersection and Union Types (1)

- *Intersection* type of a set of types S is a subtype of every set T in S and of the intersection of every subset of S
- *Union* type of a set of types S is a supertype of every set T in S and of the union of every subset of S
- Neither intersection nor union are first-class types
  - Used solely for type inferences (Chapter 20)
  - Cannot be expressed directly in programs

8.8 Intersection and Union Types (2)

- Intersection of a set of types S is equal to a named type U when any subtype of T in S and of the intersection of every subset of S is a subtype of U
- Union of a set of types S is equal to a named type U when any supertype of T in S and of the union of every superset of S is a subtype of U

```
trait S comprises {U, V} end
trait T comprises {V, W} end
trait U extends S excludes W end
trait V extends {S, T} end
trait W extends T end
```

Any subtype of S and T must be a subtype of V; V is an intersection of S and T

8.8 Intersection and Union Types (3)

Intersection types (denoted by $\cap$) possess the following properties:

- Commutativity: $T \cap U = U \cap T$.
- Associativity: $S \cap (T \cap U) = (S \cap T) \cap U$.
- Subsumption: If $S \preceq T$ then $S \cap T = S$.
- Preservation of shared subtypes: If $T \preceq S$ and $T \preceq U$ then $T \preceq S \cap U$.
- Preservation of supertype: If $S \preceq T$ then $\forall U.\ S \cap U \preceq T$.
- Distribution over union types: $S \cap (T \cup U) = (S \cap T) \cup (S \cap U)$.

8.8 Intersection and Union Types (4)

Union types (denoted by $\cup$) possess the following properties:

- Commutativity: $T \cup U = U \cup T$.
- Associativity: $S \cup (T \cup U) = (S \cup T) \cup U$.
- Subsumption: If $S \preceq T$ then $S \cup T = T$.
- Preservation of shared supertypes: If $S \preceq T$ and $U \preceq T$ then $S \cup U \preceq T$.
- Preservation of subtype: If $T \preceq S$ then $\forall U.\ T \preceq S \cup U$.
- Distribution over intersection types: $S \cup (T \cap U) = (S \cup T) \cap (S \cup U)$.

8.9 Type Aliases

Syntax:

TypeAlias ::= `type` *Id StaticParams?* = *TypeRef*

- Fortress allows names to serve as aliases for more complex type instantiations
  - All use of type aliases are expanded before type checking
  - Type aliases do not define new types nor nominal equivalence relations among types

```
type IntList = List⟦ℤ64⟧
type BinOp = Float × Float → Float
type SimpleFloat⟦nat e, nat s⟧ = DetailedFloat⟦Unity, e, s, false, false, false, false, true⟧
```

Chapter 9 Traits

- *Traits* are declared by trait declaration
- Traits define new named types
- Trait specifies a collection of *methods*
- Trait can extend others
  - Trait inherits the methods from those traits
  - Type defined by that trait is a subtype of traits it extends

9.1 Trait Declarations (1)

Syntax:

TraitDecl	::=	*TraitHeader GoInATrait*? `end`
TraitHeader	::=	*TraitMods*? `trait` *Id StaticParams*? *Extends*? *TraitClauses*?
TraitClauses	::=	*TraitClause*$^+$
TraitClause	::=	*Excludes*
	|	*Comprises*
	|	*Where*
GoInATrait	::=	*GoFrontInATrait GoBackInATrait*?
	|	*GoBackInATrait*
GoFrontInATrait	::=	*GoesFrontInATrait*$^+$
GoesFrontInATrait	::=	*AbsFldDecl*
	|	*GetterSetterDecl*
	|	*PropertyDecl*
GoBackInATrait	::=	*GoesBackInATrait*$^+$
GoesBackInATrait	::=	*MdDecl*
	|	*PropertyDecl*
TraitMods	::=	*TraitMod*$^+$
TraitMod	::=	`value` | *UniversalMod*
UniversalMod	::=	`private` | `test`
Extends	::=	`extends` *TraitTypes*
Excludes	::=	`excludes` *TraitTypes*
Comprises	::=	`comprises` *ComprisingTypes*
TraitTypes	::=	*TraitType*
	|	{ *TraitTypeList* }
TraitTypeList	::=	*TraitType*(, *TraitType*)*

9.1 Trait Declarations (2)

ComprisingTypes	::=	*TraitType*
	|	{ *ComprisingTypeList* }
ComprisingTypeList	::=	...
	|	*TraitType*(, *TraitType*)* (, ...)?
TraitType	::=	*DottedId* ([\[*StaticArgList* \])?
	|	{ *TypeRef* ↦ *TypeRef* }
	|	⟨ *TypeRef* ⟩
	|	*TypeRef* [*ArraySize*?]
	|	*TypeRef* ^ *StaticArg*
	|	*TypeRef* ^ (*ExtentRange* (× *ExtentRange*)*)
StaticArgList	::=	*StaticArg*(, *StaticArg*)*
StaticArg	::=	*Number*
	|	*Op*
	|	*true*
	|	*false*
	|	Unity
	|	dimensionless
	|	*StaticArg* + *StaticArg*
	|	*StaticArg* · *StaticArg*
	|	*StaticArg StaticArg*
	|	*StaticArg* / *StaticArg*
	|	1 / *StaticArg*
	|	*StaticArg* ^ *StaticArg*
	|	*StaticArg* per *StaticArg*
	|	*DUPreOp StaticArg*
	|	*StaticArg DUPostOp*
	|	NOT *StaticArg*
	|	*StaticArg* OR *StaticArg*
	|	*StaticArg* AND *StaticArg*
	|	*StaticArg* IMPLIES *StaticArg*
	|	*TypeRef*
	|	(*StaticArg*)
Number	::=	*IntLiteral*
ArraySize	::=	*ExtentRange*(, *ExtentRange*)*

9.1 Trait Declarations (3)

- Trait declaration:

  [modifier] trait trait_name static_params
  [extended traits]
  [excluded traits] [comprises on the trait]
  [where clause]
  {abstract_fields, getter_methods, setter_ methods}
  method_declarations
  end

- extends, excludes, comprises {trait_references}
  - If clause contains only one trait, { } may be elided
- comprises clause may include "…"
- Trait_references in comprises clause is a declared trait identifier or an abbreviated type for aggregate expressions

9.1 Trait Declarations (4)

- Every trait extends the trait Object
- extends clause every trait listed in its clause
  - If T extends U, T is a subtrait of U; U is a supertrait of T
  - Extension is transitive; if T extends U it also extends all supertraits of U
  - Extension relation is the smallest relation satisfying transitivity
  - Relation must form acyclic hierarchy rooted at trait Object

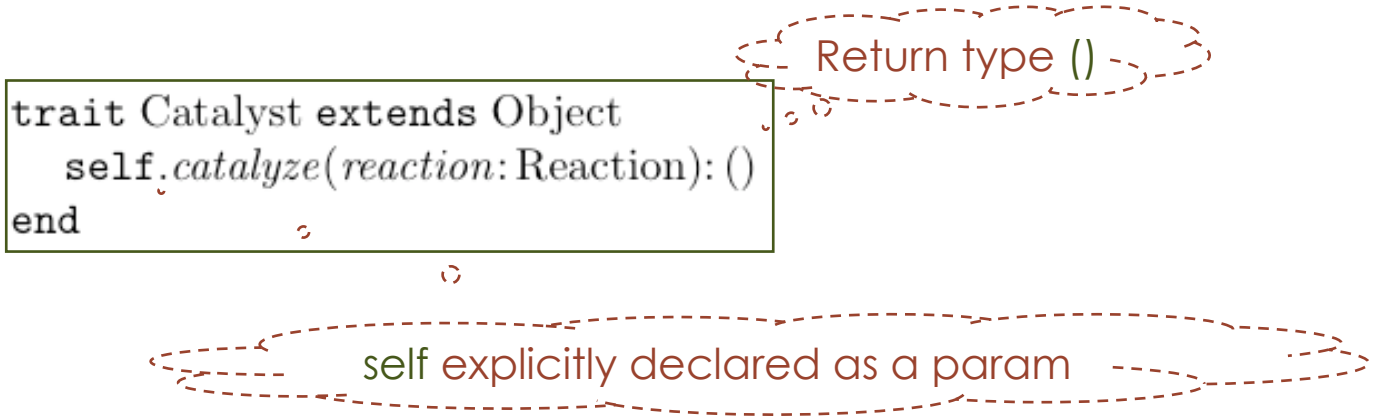
9.1 Trait Declarations (5)

- Trait T *strictly extends* U iff T extends U and T is not U
- Trait T *immediately extends* U iff T strictly extends U and there is no trait V s.t. T strictly extends V and V strictly extends U
  - U is an *immediate supertrait* of T
  - T is an *immediate subtrait* of U

9.1 Trait Declarations (6)

- Trait with excludes clause excludes every trait listed in its clause
  - If T excludes U, T and U are mutually exclusive
  - No third trait can extend them both and neither can extend the other
- If trait decl of T includes comprises clause
  - If comprises clause of T does not include "…", the trait must not be extended with immediate subtraits other than those listed in its comprises clause
  - If comprises clause of T includes "…", any subtrait of T is not exposed by API

9.1 Trait Declarations (7)



9.1 Trait Declarations (8)

```
trait Molecule comprises { OrganicMolecule, InorganicMolecule }
    mass(): Mass
end
```

Molecule can be Immediately extended by OrganicMolecule or InorganicMolecule

```
(* Not allowed! *)
trait ExclusiveMolecule extends Molecule end
```

OrganicMolecule and InorganicMolecule May be exclusive

```
trait OrganicMolecule extends Molecule excludes InorganicMolecule end
trait InorganicMolecule extends Molecule end
```

```
(* Not allowed! *)
trait InclusiveMolecule extends { InorganicMolecule, OrganicMolecule } end
```

9.2 Method Declarations (1)